

Camel

IN ACTION



SECOND EDITION

Claus Ibsen
Jonathan Anstey
FOREWORDS BY James Strachan
AND Dr. Mark Little



MANNING

contents

Cover

Camel in Action

Praise for the First Edition

Copyright

Dedication

Foreword

Foreword

Foreword to the First Edition

Preface

Acknowledgments

About This Book

Roadmap

Who should read this book

Code conventions

Source code downloads

Software requirements

Author Online

About the Authors

About the Cover Illustration

Part 1: First steps

Chapter 1: Meeting Camel

1.1 Introducing Camel

1.1.1 What is Camel?

1.1.2 Why use Camel?

1.2 Getting started

1.2.1 Getting Camel

1.2.2 Your first Camel ride

1.3 Camel's message model

1.3.1 Message

1.3.2 Exchange

1.4 Camel's architecture

1.4.1 Architecture from 10,000 feet

1.4.2 Camel concepts

1.5 Your first Camel ride, revisited

1.6 Summary

Chapter 2: Routing with Camel

2.1 Introducing Rider Auto Parts

2.2 Understanding endpoints

2.2.1 Consuming from an FTP endpoint

2.2.2 Sending to a JMS endpoint

2.3 Creating routes in Java

2.3.1 Using RouteBuilder

2.3.2 Using the Java DSL

2.4 Defining routes in XML

2.4.1 Bean injection and Spring

2.4.2 The XML DSL

2.4.3 Using Camel and Spring

2.5 Endpoints revisited

2.5.1 Sending to dynamic endpoints

2.5.2 Using property placeholders in endpoint URIs

2.5.3 Using raw values in endpoint URIs

2.5.4 Referencing registry beans in endpoint URIs

2.6 Routing and EIPs

2.6.1 Using a content-based router

2.6.2 Using message filters

2.6.3 Using multicasting

2.6.4 Using recipient lists

2.6.5 Using the wireTap method

2.7 Summary and best practices

Part 2: Core Camel

Chapter 3: Transforming data with Camel

3.1 Data transformation overview

3.2 Transforming data by using EIPs and Java

3.2.1 Using the Message Translator EIP

3.2.2 Using the Content Enricher EIP

3.3 Transforming XML

3.3.1 *Transforming XML with XSLT*

3.3.2 *Transforming XML with object marshaling*

3.4 Transforming with data formats

3.4.1 *Data formats provided with Camel*

3.4.2 *Using Camel's CSV data format*

3.4.3 *Using Camel's Bindy data format*

3.4.4 *Using Camel's JSON data format*

3.4.5 *Configuring Camel data formats*

3.5 Transforming with templates

3.5.1 *Using Apache Velocity*

3.6 Understanding Camel type converters

3.6.1 *How the Camel type-converter mechanism works*

3.6.2 *Using Camel type converters*

3.6.3 *Writing your own type converter*

3.7 Summary and best practices

Chapter 4: Using beans with Camel

4.1 Using beans the hard way and the easy way

4.1.1 *Invoking a bean from pure Java*

4.1.2 *Invoking a bean defined in XML DSL*

4.1.3 *Using beans the easy way*

4.2 Understanding the Service Activator pattern

4.3 Using Camel's bean registries

4.3.1 *JndiRegistry*

4.3.2 *SimpleRegistry*

4.3.3 *ApplicationContextRegistry*

4.3.4 *OsgiServiceRegistry and BlueprintContainerRegistry*

4.3.5 *CdiBeanRegistry*

4.4 Selecting bean methods

4.4.1 *How Camel selects bean methods*

4.4.2 *Camel's method-selection algorithm*

4.4.3 *Some method-selection examples*

4.4.4 *Potential method-selection problems*

4.4.5 *Method selection using type matching*

4.5 Performing bean parameter binding

4.5.1 *Binding with multiple parameters*

4.5.2 *Binding using built-in types*

4.5.3 *Binding using Camel annotations*

4.5.4 *Binding using Camel language annotations*

4.5.5 *Parameter binding using method name with signature*

4.6 Using beans as predicates and expressions

4.6.1 *Using beans as predicates in routes*

4.6.2 *Using beans as expressions in routes*

4.7 Summary and best practices

Chapter 5: Enterprise integration patterns

5.1 Introducing enterprise integration patterns

5.1.1 The Aggregator and Splitter EIPs

5.1.2 The Routing Slip and Dynamic Router EIPs

5.1.3 The Load Balancer EIP

5.2 The Aggregator EIP

5.2.1 Using the Aggregator EIP

5.2.2 Completion conditions for the Aggregator

5.2.3 Using persistence with the Aggregator

5.2.4 Using recovery with the Aggregator

5.3 The Splitter EIP

5.3.1 Using the Splitter

5.3.2 Using beans for splitting

5.3.3 Splitting big messages

5.3.4 Aggregating split messages

5.3.5 When errors occur during splitting

5.4 The Routing Slip EIP

5.4.1 Using the Routing Slip EIP

5.4.2 Using a bean to compute the routing slip header

5.4.3 Using an Expression as the routing slip

5.4.4 Using @RoutingSlip annotation

5.5 The Dynamic Router EIP

5.5.1 Using the Dynamic Router

5.5.2 Using the @DynamicRouter annotation

5.6 The Load Balancer EIP

5.6.1 Introducing the Load Balancer EIP

5.6.2 Using load-balancing strategies

5.6.3 Using the failover load balancer

5.6.4 Using a custom load balancer

5.7 Summary and best practices

Chapter 6: Using components

6.1 Overview of Camel components

6.1.1 Manually adding components

6.1.2 Autodiscovering components

6.2 Working with files: File and FTP components

6.2.1 Reading and writing files with the File component

6.2.2 Accessing remote files with the FTP component

6.3 Asynchronous messaging: JMS component

6.3.1 Sending and receiving messages

6.3.2 Request-reply messaging

6.3.3 Message mappings

6.4 Networking: Netty4 component

6.4.1 Using Netty for network programming

6.4.2 Using custom codecs

6.5 Working with databases: JDBC and JPA components

6.5.1 Accessing data with the JDBC component

6.5.2 Persisting objects with the JPA component

6.6 In-memory messaging: Direct, Direct-VM, SEDA, and

VM components

6.6.1 Synchronous messaging with Direct and Direct-VM

6.6.2 Asynchronous messaging with SEDA and VM

6.7 Automating tasks: Scheduler and Quartz2 components

6.7.1 Using the Scheduler component

6.7.2 Enterprise scheduling with Quartz

6.8 Working with email

6.8.1 Sending mail with SMTP

6.8.2 Receiving mail with IMAP

6.9 Summary and best practices

Part 3: Developing and testing

Chapter 7: Microservices

7.1 Microservices overview

7.1.1 Small in size

7.1.2 Observable

7.1.3 Designed for failure

7.1.4 Highly configurable

7.1.5 Smart endpoints and dumb pipes

7.1.6 Testable

7.2 Running Camel microservices

7.2.1 Standalone Camel as microservice

7.2.2 CDI Camel as microservice

7.2.3 WildFly Swarm with Camel as microservice

7.2.4 Spring Boot with Camel as microservice

7.3 Calling other microservices

7.3.1 Recommendation prototype

7.3.2 Shopping cart prototype

7.3.3 Rules and inventory prototypes

7.3.4 Rating prototype

7.3.5 Putting all the microservices together

7.4 Designing for failures

7.4.1 Using the Retry pattern to handle failures

7.4.2 Using the Retry pattern without Camel

7.4.3 Using Circuit Breaker to handle failures

7.4.4 Netflix Hystrix

7.4.5 Using Hystrix with Camel

7.4.6 Configuring Hystrix

7.4.7 Bulkhead pattern

7.4.8 Calling other microservices with fault-tolerance

7.4.9 Using Camel Hystrix with Spring Boot

7.4.10 The Hystrix dashboard

7.5 Summary and best practices

Chapter 8: Developing Camel projects

8.1 Managing projects with Maven

8.1.1 Using Camel Maven archetypes

8.1.2 Using Maven to add Camel dependencies

8.2 Using Camel in Eclipse

8.2.1 Creating a new Camel project

8.3 Debugging an issue with your new Camel project

8.4 Developing custom components

8.4.1 Setting up a new Camel component

8.4.2 Diving into the implementation

8.5 Generating components with the API component framework

8.5.1 Generating the skeleton API project

8.5.2 Configuring the camel-api-component-maven-plugin

8.5.3 Setting advanced configuration options

8.5.4 Implementing remaining functionality

8.6 Developing data formats

8.6.1 Generating the skeleton data format project

8.6.2 Writing the custom data format

8.7 Summary and best practices

Chapter 9: Testing

9.1 Introducing the Camel Test Kit

9.1.1 Using the Camel JUnit extensions

9.1.2 Using camel-test to test Java Camel routes

9.1.3 Unit testing an existing RouteBuilder class

9.2 Testing Camel with Spring, OSGi, and CDI

9.2.1 Camel testing with Spring XML

[9.2.2 Camel testing with Spring Java Config](#)

[9.2.3 Camel testing with Spring Boot](#)

[9.2.4 Camel testing with OSGi Blueprint XML](#)

[9.2.5 Camel testing with CDI](#)

[9.2.6 Camel testing with WildFly Swarm](#)

[9.2.7 Camel testing with WildFly](#)

[9.3 Using the mock component](#)

[9.3.1 Introducing the mock component](#)

[9.3.2 Unit-testing with the mock component](#)

[9.3.3 Verifying that the correct message arrives](#)

[9.3.4 Using expressions with mocks](#)

[9.3.5 Using mocks to simulate real components](#)

[9.4 Simulating errors](#)

[9.4.1 Simulating errors using a processor](#)

[9.4.2 Simulating errors using mocks](#)

[9.4.3 Simulating errors using interceptors](#)

[9.4.4 Using adviceWith to add interceptors to an existing route](#)

[9.4.5 Using adviceWith to manipulate routes for testing](#)

[9.4.6 Using weave with adviceWith to amend routes](#)

[9.5 Camel integration testing](#)

[9.5.1 Performing integration testing](#)

[9.5.2 Using NotifyBuilder](#)

9.6 Using third-party testing libraries

9.6.1 Using Arquillian to test Camel applications

9.6.2 Using Pax Exam to test Camel applications

9.6.3 Using other testing libraries

9.7 Summary and best practices

Chapter 10: RESTful web services

10.1 RESTful services

10.1.1 Understanding the principles of a RESTful API

10.1.2 Using JAX-RS with REST services

10.1.3 Using Camel in an existing JAX-RS application

10.1.4 Using camel-restlet with REST services

10.1.5 Using camel-cxf with REST services

10.2 The Camel Rest DSL

10.2.1 Exploring a quick example of the Rest DSL

10.2.2 Understanding how the Rest DSL works

10.2.3 Using supported components for the Rest DSL

10.2.4 Configuring Rest DSL

10.2.5 Using XML and JSON data formats with Rest DSL

10.2.6 Calling RESTful services using Rest DSL

10.3 API documentation using Swagger

10.3.1 Using Swagger with JAX-RS REST services

10.3.2 Using Swagger with Rest DSL

10.3.3 Documenting Rest DSL services

10.3.4 Documenting input, output, and error codes

10.3.5 Configuring API documentation

10.3.6 Using CORS and the Swagger web console

10.4 Summary and best practices

Part 4: Going further with Camel

Chapter 11: Error handling

11.1 Understanding error handling

11.1.1 Recoverable and irrecoverable errors

11.1.2 Where Camel's error handling applies

11.2 Using error handlers in Camel

11.2.1 Using the default error handler

11.2.2 The dead letter channel error handler

11.2.3 The transaction error handler

11.2.4 The no error handler

11.2.5 The logging error handler

11.2.6 Features of the error handlers

11.3 Using error handlers with redelivery

11.3.1 An error-handling use case

11.3.2 Using redelivery

11.3.3 Using DefaultErrorHandler with redelivery

11.3.4 Error handlers and scopes

11.3.5 Reusing context-scoped error handlers

11.4 Using exception policies

11.4.1 Understanding how `onException` catches exceptions

11.4.2 Understanding how `onException` works with redelivery

11.4.3 Understanding how `onException` can handle exceptions

11.4.4 Custom exception handling

11.4.5 New exception while handling exception

11.4.6 Ignoring exceptions

11.4.7 Implementing an error-handler solution

11.4.8 Bridging the consumer with Camel's error handler

11.5 Working with other error-handling features

11.5.1 Using `onWhen`

11.5.2 Using `onExceptionOccurred`

11.5.3 Using `onRedeliver`

11.5.4 Using `retryWhile`

11.6 Summary and best practices

Chapter 12: Transactions and idempotency

12.1 Why use transactions?

12.1.1 The Rider Auto Parts partner integration application

12.1.2 Setting up the JMS broker and the database

12.1.3 The story of the lost message

12.2 Transaction basics

12.2.1 Understanding Spring's transaction support

12.2.2 Adding transactions

12.2.3 Testing transactions

12.3 The Transactional Client EIP

12.3.1 Using local transactions

12.3.2 Using global transactions

12.3.3 Transaction starting from a database resource

12.3.4 Transaction redeliveries

12.3.5 Using different transaction propagations

12.3.6 Returning a custom response when a transaction fails

12.4 Compensating for unsupported transactions

12.4.1 Introducing UnitOfWork

12.4.2 Using synchronization callbacks

12.4.3 Using onCompletion

12.5 Idempotency

12.5.1 Idempotent Consumer EIP

12.5.2 Idempotent repositories

12.5.3 Clustered idempotent repository

12.6 Summary and best practices

Chapter 13: Parallel processing

13.1 Introducing concurrency

13.1.1 Running the example without concurrency

13.1.2 Using concurrency

13.2 Using thread pools

13.2.1 Understanding thread pools in Java

13.2.2 Using Camel thread pool profiles

13.2.3 Creating custom thread pools

13.2.4 Using ExecutorServiceManager

13.3 Parallel processing with EIPs

13.3.1 Using concurrency with the Threads EIP

13.3.2 Using concurrency with the Multicast EIP

13.3.3 Using concurrency with the Wire Tap EIP

13.4 Using the asynchronous routing engine

13.4.1 Hitting the scalability limit

13.4.2 Scalability in Camel

13.4.3 Components supporting asynchronous processing

13.4.4 Asynchronous API

13.4.5 Writing a custom asynchronous component

13.4.6 Potential issues when using an asynchronous component

13.5 Summary and best practices

Chapter 14: Securing Camel

14.1 Securing your configuration

14.1.1 Encrypting configuration

14.1.2 Decrypting configuration

14.2 Web service security

14.2.1 Authentication in web services

14.2.2 Authenticating web services using JAAS

14.3 Payload security

14.3.1 Digital signatures

14.3.2 Payload encryption

14.4 Transport security

14.4.1 Defining global SSL configuration

14.5 Route authentication and authorization

14.5.1 Configuring Spring Security

14.6 Summary and best practices

Part 5 : Running and managing Camel

Chapter 15: Running and deploying Camel

15.1 Starting Camel

15.1.1 How Camel starts

15.1.2 Camel startup options

15.1.3 Ordering routes

15.1.4 Disabling autostartup

15.2 Starting and stopping routes at runtime

15.2.1 Using CamelContext to start and stop routes at runtime

15.2.2 Using the Control Bus EIP to start and stop routes at runtime

15.2.3 Using RoutePolicy to start and stop routes at runtime

15.3 Shutting down Camel

15.3.1 Graceful shutdown

15.4 Deploying Camel

15.4.1 Embedded in a Java application

15.4.2 Embedded in a web application

15.4.3 Embedded in WildFly

15.5 Camel and OSGi

15.5.1 Setting up Maven to generate an OSGi bundle

15.5.2 Installing and running Apache Karaf

15.5.3 Using an OSGi Blueprint-based Camel route

15.5.4 Deploying the example

15.5.5 Using a managed service factory to spin up route instances

15.6 Camel and CDI

15.7 Summary and best practices

Chapter 16: Management and monitoring

16.1 Monitoring Camel

16.1.1 Checking health at the network level

16.1.2 Checking health level at the JVM level

16.1.3 Checking health at the application level

16.2 Using JMX with Camel

16.2.1 Using JConsole to manage Camel

16.2.2 Using JConsole to remotely manage Camel

16.2.3 Using Jolokia to manage Camel

16.3 Tracking application activity

16.3.1 Using log files

16.3.2 Using core logs

16.3.3 Using custom logging

16.3.4 Using Tracer

16.3.5 Using notifications

16.4 Managing Camel applications

16.4.1 Managing Camel application lifecycles

16.4.2 Using Jolokia and hawtio to manage Camel lifecycles

16.4.3 Using Control Bus to manage Camel

16.5 The Camel management API

16.5.1 Accessing the Camel management API using Java

16.5.2 Using Camel management API from within Camel

16.5.3 Performance statistics

16.5.4 Management-enable custom Camel components

16.5.5 Management-enable custom Java beans

16.6 Summary and best practices

Part 6: Out in the wild

Chapter 17: Clustering

17.1 Clustered HTTP

17.2 Clustered Camel routes

17.2.1 Active/passive mode

17.2.2 Active/active mode

17.2.3 Clustered active/passive mode using Hazelcast

[17.2.4 Clustered active/pассив mode using Consul](#)

[17.2.5 Clustered active/pассив mode using ZooKeeper](#)

[17.3 Clustered JMS](#)

[17.3.1 Client-side clustering with JMS and ActiveMQ](#)

[17.4 Clustered Kafka](#)

[17.4.1 Kafka consumer offset](#)

[17.4.2 Crashing a JVM with a running Kafka consumer](#)

[17.5 Clustering caches](#)

[17.5.1 Clustered cache using Hazelcast](#)

[17.5.2 Clustered cache using JCache and Infinispan](#)

[17.6 Using clustered scheduling](#)

[17.6.1 Clustered scheduling using Quartz](#)

[17.7 Calling clustered services using the Service Call EIP](#)

[17.7.1 How the Service Call EIP works](#)

[17.7.2 Service Call using static service registry](#)

[17.7.3 Service Call with failover](#)

[17.7.4 Configuring Service Call EIP](#)

[17.7.5 Service Call URI templating](#)

[17.7.6 Service Call using Spring Boot Cloud and Consul](#)

[17.8 Summary and best practices](#)

[Chapter 18: Microservices with Docker and Kubernetes](#)

[18.1 Getting started with Camel on Docker](#)

[18.1.1 Building and running Camel microservices locally](#)

18.1.2 Building and running Camel microservices using Docker

18.1.3 Building a Docker image using the Docker Maven plugin

18.1.4 Running Java microservices on Docker

18.2 Getting started with Kubernetes

18.2.1 Installing Minikube

18.2.2 Starting Minikube

18.3 Running Camel and other applications in Kubernetes

18.3.1 Running applications using kubectl

18.3.2 Calling a service running inside a Kubernetes cluster

18.3.3 Running Java applications in Kubernetes using Maven tooling

18.3.4 Java microservices calling each other in the cluster

18.3.5 Debugging Java applications in Kubernetes

18.4 Understanding Kubernetes

18.4.1 Introducing Kubernetes

18.4.2 Kubernetes architecture

18.4.3 Essential Kubernetes concepts

18.5 Building resilient Camel microservices on Kubernetes

18.5.1 Scaling up microservices

18.5.2 Using readiness and liveness probes

18.5.3 Dealing with failures by calling services in Kubernetes

18.6 Testing Camel microservices on Kubernetes

18.6.1 Setting up Arquillian Cube

18.6.2 Writing a basic unit test using Arquillian Cube

18.6.3 Running Arquillian Cube tests on Kubernetes

18.6.4 Writing a unit test that calls a Kubernetes service

18.7 Introducing fabric8, Helm, and OpenShift

18.7.1 fabric8

18.7.2 Kubernetes Helm

18.7.3 OpenShift

18.8 Summary and best practices

Chapter 19: Camel tooling

19.1 Camel editors

19.1.1 JBoss Fuse Tooling

19.1.2 Apache Camel IDEA plugin

19.1.3 Camel validation using Maven

19.2 Camel Catalog: the information goldmine

19.3 hawtio: a web console for Camel and Java applications

19.3.1 Understanding hawtio functionality

19.3.2 Debugging Camel routes using hawtio

19.4 Summary and best practices

Chapter 20: Reactive Camel

20.1 Using Reactive Streams with Camel

20.1.1 Reactive Streams API

20.1.2 Reactive flow control with back pressure

20.1.3 First steps with Reactive Streams

20.1.4 Using Camel with Reactive Streams

20.1.5 Controlling back pressure from the producer side

20.1.6 Controlling back pressure from the consumer side

20.2 Using Vert.x with Camel

20.2.1 Building a football simulator using Vert.x

20.2.2 Using Camel together with Vert.x

20.2.3 Summary of using Camel with Vert.x for microservices

20.3 Summary and best practices

Chapter 21: Camel and the IoT

21.1 The Internet of Things shopping list

21.1.1 Raspberry Pi

21.1.2 SD card for Raspberry Pi

21.1.3 Power bank for Raspberry Pi

21.1.4 Camera for Raspberry Pi

21.1.5 TI SensorTag

21.2 The Internet of Things architecture

21.3 Why Camel is the right choice for the IoT

21.3.1 Components

21.3.2 Data formats

21.3.3 Redelivery

21.3.4 Throttling

21.3.5 Content-based routing

21.3.6 Client-side load balancing

21.3.7 Control bus

21.4 Gateway-to-data-center connectivity

21.4.1 Understanding the architecture

21.4.2 Choosing a protocol

21.5 Camel and Eclipse Kura

21.5.1 Starting Kura in emulator mode

21.5.2 Defining Camel routes using the Kura web UI

21.5.3 Next steps with Camel and Kura

21.6 Next steps with Camel and the IoT

LWM2M and the Eclipse Leshan project

Eclipse Hono

21.7 Summary

Appendix A: Simple, the expression language

A.1 Introducing Simple

A.2 Syntax

A.3 Built-in variables

A.4 Built-in functions

A.5 Built-in file variables

A.6 Built-in operators

A.6.1 Combining expressions

A.7 The OGNL feature

A.8 Using Simple from custom Java code

Summary

Appendix B: The Camel community

Apache Camel website

JIRA, mailing lists, Gitter, and IRC

Camel at GitHub

Camel at Stack Overflow

Commercial Camel offerings

Camel tooling

Camel-extra project

Becoming a Camel committer

Videos

Other resources

Index

Camel in Action

Second Edition

CLAUS IBSEN
JONATHAN ANSTEY



MANNING
Shelter Island

Praise for the First Edition

I highly recommend this book. It kicks ass!

—James Strachan, cofounder of Apache Camel

Strikes the right balance between core concepts and running code.

—Gregor Hohpe, coauthor of *Enterprise Integration Patterns*
Great content from the source developers.

—Domingo Suarez Torres, SynergyJ

Comprehensive guide to enterprise integration with Camel.

—Gordon Dickens, Chariot Solutions

A deep book... with great examples.

—Jeroen Benckhuijsen, Atos Origin

A tech library essential!

—Mick Knutson, BASE Logic

If you want to get a good understanding of what Camel can do and how Camel does it, this book should be your first choice.

—Willem Jiang, Progress Software

This is my go-to book when using Camel in several real-world commercial projects. Highly recommended!

—Michael Nash, Point2 Technologies

Provides developers an excellent treatise on building integration applications.

—Bruce Snyder, SpringSource

A must-have for solving integration issues.

—Tijs Rademakers, Atos Origin

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department

Manning Publications Co.

20 Baldwin Road

PO Box 761

Shelter Island, NY 11964

Email: orders@manning.com

©2018 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

Manning Publications Co.

20 Baldwin Road

PO Box 761

Shelter Island, NY 11964

Development editor: Cynthia Kane

Technical development editor: Alain Couniot

Project editors: Kevin Sullivan and Janet Vail

Copyeditor: Sharon Wilkey

Proofreader: Corbin Collins

Technical proofreader: John Guthrie

Typesetter: Happenstance Type-O-Rama

Cover designer: Marija Tudor

ISBN 9781617292934

Printed in the United States

1 2 3 4 5 6 7 8 9 10 - EBM - 23 22 21 20 19 18

$$\mathscr{T}$$

$$c\,$$

$${\mathcal O}$$

$${\mathcal O}$$

$$t_{\mathrm{max}}=100$$

$$m_{\rm c}$$

$$\mathbb{H}^{\ast}$$

$$\mathfrak{e}_+$$

foreword

When I first saw Gregor Hohpe and Bobby Woolf's *Enterprise Integration Patterns* book, I knew the patterns and ideas from the book would change the face of integration software. I figured it was time for a simple, easy-to-use implementation of those patterns, so I created the first version of Apache Camel over 10 years ago now. I had no idea how large and vibrant the Apache Camel community would become, how rich and diverse its capabilities would become, how great the number of patterns supported would be, nor how many connectors, languages, and tools would be integrated.

When Camel was still a relative toddler, along came Claus Ibsen, who soon became Mr. Camel—the main leader, maintainer, and evangelist behind Apache Camel. I can't think of a better set of authors to write about Apache Camel than Claus “Mr. Camel” Ibsen and our fellow colleague and long-time Camel committer, Jonathan Anstey.

I loved the first edition of the book and am amazed at how much has changed for the second edition from the Apache Camel project and ecosystem, in particular how much better Camel works in the cloud. I highly recommend that anyone with even a passing interest in Apache Camel buy this book—and please, enjoy the ride and don't get the hump! :)

SENIOR ARCHITECT, CLOUDBEES JAMES STRACHAN
ORIGINAL AUTHOR OF APACHE CAMEL

foreword

The open source Apache Camel project has been at the forefront of the widespread adoption of Enterprise Integration Patterns on the JVM for many years. In fact it's been so popular that some developers of other popular programming languages have cited it as a strong influence when they've implemented similar efforts. Apache Camel's easy, intuitive, and extensible approach has made it possible for even novice developers to produce reliable solutions to complex problems in a realistic period of time. The vibrant open source community of contributors and users has helped evolve the project into new areas such as the cloud, mobile, and the Internet of Things (IoT). This positive feedback loop looks strong. Innovation continues on a daily basis led by a number of key contributors, most notably the authors of this book, Claus and Jon.

I first heard of Claus and Jon when I became an early user of Apache Camel in a previous role. Their programming style was clear and concise, matched only by their patience, ability to communicate complex concepts at all levels, and their insatiable thirst to learn from the community of Apache Camel users and contributors in order to continually evolve and grow the project. Fast-forward a few years, and I was able to work much more closely with them and the rest of the Fuse team when Red Hat acquired FuseSource. I've learned that my initial love of Apache Camel was not misplaced, and we've seen huge success internally and externally with it.

In this revised edition of their incredibly popular book, Claus and Jon stay true to the original formula that helped make the first version so approachable: they know their subject better than most and obviously like to help others to learn and gain that same level of understanding. The book has been structured well and encourages you to jump around and across topics, dipping in

and out where it helps you to accomplish your task. Claus and Jon are great writers too, relying on figures and diagrams where needed, with lots of code snippets and worked examples, which are always necessary in good technical books.

As I mentioned, Apache Camel has seen a lot of changes over the years, as has the industry, and the authors manage to reflect this within the updated edition. There is coverage of how the project is used within IoT, for instance. But probably one of the biggest changes in the software landscape relevant to Apache Camel since the book's first edition is the evolution of service-oriented architectures (SOAs) toward smaller-grained services known as *microservices*, as well as a move to the cloud with technologies such as Docker and Kubernetes. Claus and Jon manage to cover these relatively new and fast-moving topics, showing how Apache Camel should remain a key part of the software architect's toolbox.

TECHNICAL DIRECTOR OF JBOSS DR. MARK LITTLE
VICE PRESIDENT OF ENGINEERING, RED HAT

foreword to the first edition

Languages are a critical aspect of software development. They give us the vocabulary to express what a program should do. They force us to encode our requirements in precise and unambiguous terms. Lastly, they enable the sharing of knowledge between developers. No, I'm not talking about Java, Haskell, or PL/1. I'm talking about the languages we use to communicate from human to human, from developer to developer, or from end user to product manager. For a long time, the world of enterprise integration (or EAI, as it was commonly known in the “dark ages of integration”) lacked such a vocabulary. Each vendor offered a proprietary solution that not only failed to integrate at a technical level with other vendors’ offerings, but also used a different language to describe the main components and their functions. This caused confusion and was also a key inhibitor to creating a community of developers that could span the vast space of enterprise integration. Each “tribe” was essentially held hostage by the language bestowed upon them. Ironically, integration developers were faced with the same Tower of Babel problem that their software was designed to solve!

Establishing a common vocabulary that enables knowledge sharing and collaboration was the key motivator for us to write the *Enterprise Integration Patterns* book. Each of the 65 patterns has a descriptive name that represents the solution to a design challenge in the integration space. Besides supporting effective communication, this vocabulary also raises the level of abstraction at which we can describe integration problems and solutions.

A shared vocabulary is a big step forward, but a giant step we could not imagine at the time was that our language would spur the development of a whole family of open source messaging and

enterprise service bus (ESB) products. These tools embrace the EIP vocabulary by implementing many patterns directly in the platform. With Apache Camel, a Splitter pattern translates directly into a “split” element in the Camel DSL. We couldn’t have wished for a more direct translation of the pattern language into an implementation platform.

Claus and Jon bring the saga to a grand finale by showing us how to use the Camel pattern language to compose real-life messaging solutions. In doing so, they not only cover fundamental concepts like routing and transformation, but also dig into often-neglected parts of the development process, including testing, monitoring, and deploying. They find the right balance of the pattern language, Camel core concepts, and running code to help you build easy-to-understand and robust messaging solutions.

COAUTHOR OF *ENTERPRISE INTEGRATION PATTERNS* GREGOR HOHPE
WWW.EAIPATTERNS.COM

preface

Developers who have done integration work know what a difficult task it can be. IT systems may not have been designed to be accessible from other systems, and if they were designed for interoperability, they may not speak the protocol you need. As a developer, you end up spending a considerable amount of time working with the plumbing of the integration protocols to open up the IT systems to the outside world.

In *Enterprise Integration Patterns*, Gregor Hohpe and Bobby Woolf gave us a standard way to describe, document, and implement complex integration problems. Developers and architects alike can use this common language and catalog of solutions to tackle their integration problems. But although Hohpe and Woolf gave us the theory, the industry still needed an open source implementation of the book.

In his foreword, James Strachan explains how Apache Camel came to life. In the beginning of 2007, James created Apache Camel as an implementation of the EIP book, and by summer, version 1.0 was released.

Apache Camel is an integration framework with the main goal of making integration easier. It implements many of the EIP patterns and allows you to focus on solving business problems, freeing you from the burden of plumbing. Using connectivity components has never been easier, because you don't have to implement JMS message listeners, FTP clients, or deal with the complexity of integrating with the massive set of SalesForce APIs and services. Camel also has great support for converting data between protocols, abstracting away lower-level raw details of such things as HTTP requests. All this is taken care of by Camel, which makes mediation and routing as easy as writing a few lines of Java code or using XML.

Since its creation Apache Camel has become very popular, and today it has an ever-growing community. As with many open source projects that become popular, a logical next step was for someone to write a book about it, and so we did. In 2008, when we started our journey on writing the first edition of this book, which was published in late 2010. After the success of the first edition, Michael Stephens from Manning got in touch with us again in March 2015 to discuss our next Camel book. We went down the rabbit hole again and in July 2015 signed up for writing *Camel in Action*, Second Edition.

Writing this second edition has been a long journey, proven by the fact that we started over two years ago! This extended time was partly due to us both being much busier in our personal lives, but also you may have noticed that the second edition is nearly twice as long as the first. We just couldn't stop writing! Although this may make the print book a little uncomfortable to hold, it also means you get more in-depth coverage of Camel than ever before. It's a good trade-off, we think! Despite the length, we've managed to keep up with the fast-moving Camel project. This book uses the latest Camel release at the time of writing (Camel 2.20.1).

We hope this book proves to be of great value to you and helps you prosper in the Camel community.

SENIOR PRINCIPAL SOFTWARE ENGINEER, RED HAT CLAUS IBSEN
ENGINEERING MANAGER, RED HAT JONATHAN ANSTEY

acknowledgments

We first want to thank Cynthia Kane, Kevin Sullivan, and Janet Vail, our development and project editors at Manning, who helped steer the project and ensure progress. We'd also like to thank our awesome proofreader, Corbin Collins, for turning our writing into an enjoyable reading experience. A big thank you to Susan Harkins, who helped us with our final review, ensuring our book meets the high standard you'd expect from a Manning title. The greater Manning team deserves kudos as well; they've made for a very pleasant writing experience over the past two years. We'd also like to thank Michael Stephens for getting in touch with us again and pitching the idea of a second edition. Special thanks to Alain Couniot for being our technical proofreader, catching those bugs we missed, and helping improve the source code for the book.

Big thanks to our team of reviewers, who provided invaluable feedback during various stages of the book's development: Andrea Barisone, Ethien Daniel Salinas Dominquez, Fabrizio Cucci, Gregor Zurowski, Grzegorz Grzybek, Ivan Brencsics, José Diaz, Mark Stofferahn, Philippe Van Bergen, Phillip A. Sorensen, Rambabu Posa, Rick Wagner, Tobias Kilian, Werner Aernouts, and Yan Guo.

We're also very grateful for all the help we received from people in our circles who volunteered to review the material during the two years of development. A big thank you to the following: Antoine Dessaigne, Aurélien Pupier, Bilgin Ibryam, Christian Posta, Clement Escoffier, Joseph Kampf, Kevin Earls, Lars Heinemann, Luca Burgazzoli, Nicola Ferraro, and Scott Cranton.

The accompanying source code for any technical book plays a vital role to readers in allowing them to try the examples and

experiment—to learn by doing. We want to thank the following who have helped by contributing to the source code: Aurélien Pupier, Babak Vahdat, Christoph Deppisch, Grzegorz Grzybek, Kevin Earls, Luca Burgazzoli, Morten Erik Banzon, Ryota Sato, Scott Cranton, and Willem Jiang.

We also wish to thanks all our readers of the first edition who were so kind to submit errata. We've made sure to fix those items for this second edition.

Thanks to Henryk Konsek for being our guest author of chapter 21, which is all about using Camel with the IoT (Internet of Things).

We'd like to thank Gregor Hohpe, James Strachan, and Mark Little for writing the forewords to our book. Gregor's book *Enterprise Integration Patterns* has been one of our favorite tech books for years now, so it's an honor to have Gregor on board to write a foreword. Without the EIP book, Apache Camel would look a lot different than it does today, if it existed at all. James Strachan is an inspiration to many developers out there—including us. He's founded tons of successful open source projects; Camel is just one of them. If James hadn't decided to create Camel, we wouldn't be writing this book. So, again, thanks!

Mark Little at Red Hat has been very supportive of our work on the JBoss Fuse team, and we want to thank Mark for believing in us and Apache Camel.

A warm thank you goes to our manager Aileen Cunningham at Red Hat, who has been supportive of our work and given us company time to focus on finishing this book.

Finally, we'd like to give a big warm thank you to the Camel community. Without them, the Apache Camel project wouldn't be as successful as it is today. And without that success, both of us would have different kinds of jobs today, which wouldn't involve hacking on Camel on a daily basis.

CLAUS

Writing this book the second time was an emotional rollercoaster. My 12-year relationship ended, and I moved from Sweden back to my home country of Denmark. I want to thank my parents for openly welcoming me back home and allowing me to live under their roof while my new condo was being renovated. I enjoyed every day I spent together with my father until his sudden passing on November 4, 2017. Father, I love you and I will see you in heaven when it's my turn. I will stay strong and look after Mother.

JON

I would like to thank my amazing wife, Lisa, for the patience, support, and encouragement I needed throughout the writing of this book. We took way longer than expected this time, and so the burden on you was heavy. I truly appreciate all you do, hon. To Georgia, my beautiful daughter, and Jake, my superman heart warrior: thank you for cheering me up each and every day. Love you guys!!

about this book

Apache Camel exists because integration is hard and Camel's creators wanted to make things easier for users. Camel's online documentation serves as a reference for its many features and components. This book aims to guide readers through those features, starting with the simple points and building up to advanced Camel usage by the end of the book. Throughout the book, Camel's features are put into action in real-life scenarios.

Roadmap

The book is divided into six parts:

- *Part 1*—First steps
- *Part 2*—Core Camel
- *Part 3*—Developing and testing
- *Part 4*—Going further with Camel
- *Part 5*—Running and managing Camel
- *Part 6*—Out in the wild

Part 1 starts off simply, by introducing you to Camel's core functionality and concepts, and it presents some basic examples:

- Chapter 1 introduces you to Camel and explains what Camel is and where it fits into the bigger enterprise software picture. You'll also start learning the concepts and terminology of Camel.
- Chapter 2 covers Camel's main feature: message routing. Java DSL and XML DSL are covered, as are several Enterprise Integration Patterns (EIPs). EIPs are basically canned solutions to integration problems.

Part 2 builds on the foundation of part 1 and covers the core features of Camel. You'll need many of these features when using Camel:

- Chapter 3 explains how Camel can help you transform your data to different formats while it's being routed.
- Chapter 4 takes a look at how you can use Java beans in Camel.
- Chapter 5 explores in depth the most powerful and complex EIPs.
- Chapter 6 covers the most frequently used components from Camel's large selection.

Having absorbed all the core concepts of Camel, you're ready to learn how to develop and test your Camel applications in part 3:

- Chapter 7 is an extensive chapter that teaches you all about how to use Camel with microservices. You'll find details on making the most of Camel with Spring Boot, WildFly Swarm, and other popular microservice containers.
- Chapter 8 explains how to create new Camel projects, which could be Camel applications, custom components, or data formats. This chapter doesn't require much additional Camel knowledge, so you could read it right after part 1.
- Chapter 9 looks at the testing facilities shipped with Camel. You can use these features for testing your own Camel applications or applications based on other stacks.
- Chapter 10 provides in-depth coverage of using RESTful services with Camel, including Camel's Rest DSL, and you'll see how to document your APIs using Swagger with Camel.

In part 4 we cover topics that are useful when you've gained a better understanding of Camel from earlier chapters:

- Chapter 11 covers all of Camel's error-handling features. Error handling is one of the more complex issues you'll have to deal with, so make sure to read this chapter.

- Chapter 12 explains how you can use transactions in your Camel applications.
- Chapter 13 discusses how to deal with concurrency and scalability in your Camel applications.
- Chapter 14 covers how to secure your Camel applications.

Part 5 is a two-chapter part that focuses on the deployment and management aspects of Camel, so you can run your Camel applications the best way possible in production:

- Chapter 15 talks about the many ways to reliably start and stop Camel. Deployment to several of the most popular containers is also discussed.
- Chapter 16 covers how to manage and monitor Camel applications—including, among other things, how to read the Camel logs and how to control Camel with JMX.

The last part is where we take you the extra mile and off the beaten track and show you a wide variety of things you can do with Camel:

- Chapter 17 covers the important topic of how to cluster your Camel applications.
- Chapter 18 discusses Camel, containers, and the cloud, and you'll learn how to containerize Camel to run with Docker and Kubernetes.
- Chapter 19 covers the most popular Camel tooling that comes out of the box and what's available from third parties on the internet.

The appendixes at the end of the book contain useful reference material on the Simple expression language and the Camel community.

There are also two bonus chapters available online at www.manning.com/books/camel-in-action-second-edition:

- Chapter 20 gives an introduction to Reactive systems and the

Reactive Streaming API, along with information on how to use them with Camel. You'll also find an introduction to Vert.X, a popular Reactive toolkit that works well with Camel.

- Chapter 21, written by Henryk Konsek, introduces the Internet of Things (IoT) and covers how you can integrate IoT devices with Camel.

Who should read this book

We wrote this book primarily for developers who have found the online Camel documentation lacking and need a guidebook that explains things in a more detailed and organized way. Although we mainly targeted existing Camel users, *Camel in Action* is a great way to start learning about Camel. Experienced engineers and architects are also encouraged to read this book, as it explains advanced Camel concepts that you just can't find elsewhere. Test and Q&A engineers will find Camel and this book useful as a means of driving tests that require communication with various transports and APIs. System administrators may also find the management, monitoring, and deployment topics of great value.

Camel's features are focused on the enterprise business community and its needs, but it's also a general and very useful integration toolkit. Any Java developer who needs to send a message somewhere will probably find Camel and this book useful.

Code conventions

The code examples in this book are abbreviated in the interest of space. In particular, some of the namespace declarations in the XML configurations and package imports in Java classes have been omitted. We encourage you to use the source code when working with the examples. The line lengths of some of the examples exceed the page width, and in those cases the line continuation arrow (→) is used to indicate that a line has been

wrapped for formatting.

All source code in listings or in text is in a fixed-width font like this to separate it from ordinary text. Code annotations accompany many of the listings, highlighting important concepts. In some cases, numbered bullets link to explanations that follow the listing.

Source code downloads

The source code for the examples in this book is available online at GitHub: <https://github.com/camelinaction/camelinaction2>.

Software requirements

The following software is required to run the examples:

- JDK 8
- Maven 3.5 or better
- Apache Camel 2.20.1 or better

Apache Camel itself can be downloaded from its official website: <http://camel.apache.org/download.html>.

All the examples can be run using Maven. Chapter 1 shows you how to get started with Maven and run the examples.

Author Online

The purchase of *Camel in Action* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/books/camel-in-action-second-edition. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of

conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors



CLAUS IBSEN has worked as a software engineer and architect for more than 20 years. He has often worked with integration in various forms, from integrating with legacy systems on AS/400s to building custom in-house integration frameworks. Claus has designed and architected a large solution for custom clearance for the district of Shanghai, China. He tracks the trends in the open source integration space, and it led him to Apache Camel in late 2007.

He became a committer in March 2008. In the beginning of 2009, he joined FuseSource as full-time developer on Camel and has not looked back since. He currently holds a position as senior principal software engineer at Red Hat, working as project lead on Apache Camel.

Claus also works on other open source projects related to integration, such as JBoss Fuse, Apache ActiveMQ, fabric8, hawtio, Syndesis, and Vert.X. Claus is a regular speaker at conferences, so you may have the chance to meet him face to face.

In his spare time Claus is a keen runner of half marathons. (Thank you, Jonas, for pushing me to run longer distances.) He has plans to go the full distance. He also enjoys traveling, and if the work-travel isn't sufficient he may consider going on leave to become a backpacker once again.

Claus lives in his hometown of Esbjerg, Denmark.

JONATHAN ANSTEY is a software engineer with varied experience in



manufacturing control systems, build infrastructure, and enterprise integration. He got involved in the Apache Camel project in early 2008 and hasn't looked back since. Most recently, Jon has been overseeing maintenance for Apache Camel and other open source projects as an engineering manager at Red Hat, Inc.

When Jon isn't hacking on Camel, he likes to spend time with his wife and two kids in Paradise, Newfoundland.

about the cover illustration

The illustration on the cover of *Camel in Action* bears the caption “A Bedouin” and is taken from a collection of costumes of the Ottoman Empire published on January 1, 1802, by William Miller of Old Bond Street, London. The title page is missing from the collection, and we’ve so far been unable to track it down. The book’s table of contents identifies the figures in both English and French, and each illustration also bears the names of two artists who worked on it, both of whom would no doubt be surprised to find their art gracing the front cover of a computer programming book 200 years later.

The collection was purchased by a Manning editor at an antiquarian flea market in the “Garage” on West 26th Street in Manhattan. The seller was an American based in Ankara, Turkey, and the transaction took place just as he was packing up his stand for the day. The Manning editor didn’t have on his person the substantial amount of cash that was required for the purchase, and a credit card and check were both politely turned down. With the seller flying back to Ankara that evening, the situation seemed hopeless. What was the solution? It turned out to be nothing more than an old-fashioned verbal agreement sealed with a handshake. The seller proposed that the money be transferred to him by wire, and the editor walked out with the bank information on a piece of paper and the portfolio of images under his arm. Needless to say, we transferred the funds the next day, and we remain grateful and impressed by this unknown person’s trust in one of us. It recalls something that might have happened a long time ago.

The pictures from the Ottoman collection, like the other illustrations that appear on Manning’s covers, bring to life the richness and variety of dress customs of two centuries ago. They recall the sense of isolation and distance of that period—and of

every other historic period except our own hyperkinetic present. Dress codes have changed since then certainly, and the diversity by region, so rich at the time, has faded away. It's now often hard to tell the inhabitant of one continent from that of another. Perhaps, viewed optimistically, we've traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and, yes, the *fun* of the computer business with book covers based on the rich diversity of regional life as it was two centuries ago, brought back to life by the pictures from this collection.

Part 1

First steps

Apache Camel is an open source integration framework that aims to make integrating systems easier. In the first chapter of this book, we'll introduce you to Camel and show you how it fits into the bigger enterprise software picture. You'll also learn the concepts and terminology of Camel.

Chapter 2 focuses on message routing, one of Camel's most important features. Camel has two main ways of defining routing rules: the Java-based domain specific language (DSL) and the XML configuration format. In addition to these route-creation techniques, we'll show you how to design and implement solutions to integration problems using enterprise integration patterns (EIPs) and Camel.

1

Meeting Camel

This chapter covers

- An introduction to Camel
- Camel’s main features
- Your first Camel ride
- Camel’s architecture and concepts

Building complex systems from scratch is a costly endeavor, and one that’s almost never successful. An effective and less risky alternative is to assemble a system like a jigsaw puzzle from existing, proven components. We depend daily on a multitude of such integrated systems, making possible everything from phone communications, financial transactions, and health care to travel planning and entertainment.

You can’t finalize a jigsaw puzzle until you have a complete set of pieces that plug into each other simply, seamlessly, and robustly. That holds true for system integration projects as well. But whereas jigsaw puzzle pieces are made to plug into each other, the systems we integrate rarely are. Integration frameworks aim to fill this gap. As a developer, you’re less concerned about how the system you integrate works and more focused on how to interoperate with it from the outside. A good integration framework provides simple, manageable abstractions for the complex systems you’re integrating and the “glue” for

plugging them together seamlessly.

Apache Camel is such an integration framework. In this book, we'll help you understand what Camel is, how to use it, and why we think it's one of the best integration frameworks out there. This chapter starts off by introducing Camel and highlighting some of its core features. We'll then present the Camel distribution and explain how to run the Camel examples in the book. We'll round off the chapter by bringing core Camel concepts to the table so you can understand Camel's architecture.

Are you ready? Let's meet Camel.

1.1 Introducing Camel

Camel is an integration framework that aims to make your integration projects productive and fun. The Camel project was started in early 2007 and now is a mature open source project, available under the liberal Apache 2 license, with a strong community.

Camel's focus is on simplifying integration. We're confident that by the time you finish reading these pages, you'll appreciate Camel and add it to your must-have list of tools.

This Apache project was named *Camel* because the name is short and easy to remember. Rumor has it the name may be inspired by the Camel cigarettes once smoked by one of the founders. At the Camel website, a FAQ entry (<http://camel.apache.org/why-the-name-camel.html>) lists other lighthearted reasons for the name.

1.1.1 WHAT IS CAMEL?

At the core of the Camel framework is a routing engine—or more precisely, a routing-engine builder. It allows you to define your own routing rules, decide from which sources to accept messages, and determine how to process and send those messages to other destinations. Camel uses an integration

language that allows you to define complex routing rules, akin to business processes. As shown in [Figure 1.1](#), Camel forms the glue between disparate systems.

One of the fundamental principles of Camel is that it makes no assumptions about the type of data you need to process. This is an important point, because it gives you, the developer, an opportunity to integrate any kind of system, without the need to convert your data to a canonical format.



[Figure 1.1 Camel is the glue between disparate systems.](#)

Camel offers higher-level abstractions that allow you to interact with various systems by using the same API regardless of the protocol or data type the systems are using. Components in Camel provide specific implementations of the API that target different protocols and data types. Out of the box, Camel comes with support for more than 280 protocols and data types. Its extensible and modular architecture allows you to implement and seamlessly plug in support for your own protocols, proprietary or not. These architectural choices eliminate the need for unnecessary conversions and make Camel not only faster but also lean. As a result, it's suitable for embedding into other projects that require Camel's rich processing capabilities. Other open source projects, such as Apache ServiceMix, Karaf, and ActiveMQ, already use Camel as a way to carry out integration.

We should also mention what Camel isn't. Camel isn't an enterprise service bus (ESB), although some call Camel a lightweight ESB because of its support for routing, transformation, orchestration, monitoring, and so forth. Camel doesn't have a container or a reliable message bus, but it can be deployed in one, such as the previously mentioned Apache

ServiceMix. For that reason, we prefer to call Camel an *integration framework* rather than an *ESB*.

If the mere mention of ESBs brings back memories of huge, complex deployments, don't fear. Camel is equally at home in tiny deployments such as microservices or internet-of-things (IoT) gateways.

To understand what Camel is, let's take a look at its main features.

1.1.2 WHY USE CAMEL?

Camel introduces a few novel ideas into the integration space, which is why its authors decided to create Camel in the first place. We'll explore the rich set of Camel features throughout the book, but these are the main ideas behind Camel:

- Routing and mediation engine
- Extensive component library
- Enterprise integration patterns (EIPs)
- Domain-specific language (DSL)
- Payload-agnostic router
- Modular and pluggable architecture
- Plain Old Java Object (POJO) model
- Easy configuration
- Automatic type converters
- Lightweight core ideal for microservices
- Cloud ready
- Test kit
- Vibrant community

Let's dive into the details of each of these features.

ROUTING AND MEDIATION ENGINE

The core feature of Camel is its routing and mediation engine. A *routing engine* selectively moves a message around, based on the route's configuration. In Camel's case, routes are configured with a combination of enterprise integration patterns and a domain-specific language, both of which we'll describe next.

EXTENSIVE COMPONENT LIBRARY

Camel provides an extensive library of more than 280 components. These components enable Camel to connect over transports, use APIs, and understand data formats. Try to spot a few technologies that you've used in the past or want to use in the future in [figure 1.2](#). Of course, it isn't possible to discuss all of these components in the book, but we do cover about 20 of the most widely used. Check out the index if you're interested in a particular one.



Figure 1.2 Connect to just about anything! Camel supports more than 280 transports, APIs, and data formats.

ENTERPRISE INTEGRATION PATTERNS

Although integration problems are diverse, Gregor Hohpe and Bobby Woolf noticed that many problems, and their solutions are quite similar. They cataloged them in their book *Enterprise Integration Patterns* (Addison-Wesley, 2003), a must-read for any integration professional (www.enterpriseintegrationpatterns.com). If you haven't read it,

we encourage you to do so. At the very least, it'll help you understand Camel concepts faster and easier.

The enterprise integration patterns, or EIPs, are helpful not only because they provide a proven solution for a given problem, but also because they help define and communicate the problem itself. Patterns have known semantics, which makes communicating problems much easier. Camel is heavily based on EIPs. Although EIPs describe integration problems and solutions and provide a common vocabulary, the vocabulary isn't formalized. Camel tries to close this gap by providing a language to describe the integration solutions. There's almost a one-to-one relationship between the patterns described in *Enterprise Integration Patterns* and the Camel DSL.

DOMAIN-SPECIFIC LANGUAGE

At its inception, Camel's domain-specific language (DSL) was a major contribution to the integration space. Since then, several other integration frameworks have followed suit and now feature DSLs in Java, XML, or custom languages. The purpose of the DSL is to allow the developer to focus on the integration problem rather than on the tool—the programming language. Here are some examples of the DSL using different formats and staying functionally equivalent:

- Java DSL

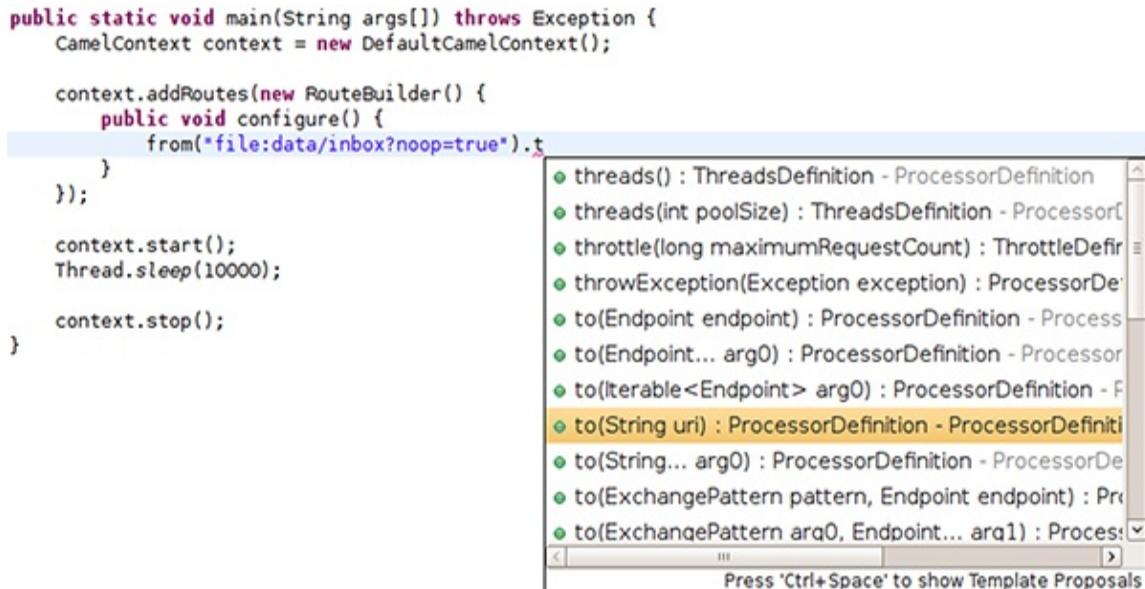
```
from("file:data/inbox").to("jms:queue:order");
```

- XML DSL

```
<route>
  <from uri="file:data/inbox"/>
  <to uri="jms:queue:order"/>
</route>
```

These examples are real code, and they show how easily you can route files from a folder to a Java Message Service (JMS) queue. Because there's a real programming language underneath, you can use the existing tooling support, such as code completion

and compiler error detection, as illustrated in figure 1.3.



The screenshot shows a Java code editor in the Eclipse IDE. A tooltip window is open over some Camel DSL code, listing various methods available for the current context. The method 'to(String uri)' is highlighted in orange, indicating it is the currently selected suggestion. Other visible suggestions include 'threads()', 'threads(int poolSize)', 'throttle(long maximumRequestCount)', 'throwException(Exception exception)', 'to(Endpoint endpoint)', 'to(Endpoint... arg0)', 'to(Iterable<Endpoint> arg0)', 'to(String uri)', 'to(String... arg0)', 'to(ExchangePattern pattern, Endpoint endpoint)', and 'to(ExchangePattern arg0, Endpoint... arg1)'. The tooltip also includes a note at the bottom: 'Press 'Ctrl+Space' to show Template Proposals'.

```
public static void main(String args[]) throws Exception {
    CamelContext context = new DefaultCamelContext();

    context.addRoutes(new RouteBuilder() {
        public void configure() {
            from("file:data/inbox?noop=true").t
        }
    });
}

context.start();
Thread.sleep(10000);

context.stop();
}
```

Figure 1.3 Camel DSLs use real programming languages such as Java, so you can use existing tooling support.

Here you can see how the Eclipse IDE's autocomplete feature can give you a list of DSL terms that are valid to use.

PAYLOAD-AGNOSTIC ROUTER

Camel can route any kind of payload; you aren't restricted to carrying a normalized format such as XML payloads. This freedom means you don't have to transform your payload into a canonical format to facilitate routing.

MODULAR AND PLUGGABLE ARCHITECTURE

Camel has a modular architecture, which allows any component to be loaded into Camel, regardless of whether the component ships with Camel, is from a third party, or is your own custom creation. You can also configure almost anything in Camel. Many of its features are pluggable and configurable—anything from ID generation, thread management, shutdown sequencer, stream caching, and whatnot.

POJO MODEL

Java beans (or Plain Old Java Objects, POJOs) are considered first-class citizens in Camel, and Camel strives to let you use beans anywhere and anytime in your integration projects. In many places, you can extend Camel's built-in functionality with your own custom code. Chapter 4 has a complete discussion of using beans within Camel.

EASY CONFIGURATION

The *convention over configuration* paradigm is followed whenever possible, which minimizes configuration requirements. In order to configure endpoints directly in routes, Camel uses an easy and intuitive URI configuration.

For example, you could configure a Camel route starting from a file endpoint to scan recursively in a subfolder and include only .txt files, as follows:

```
from("file:data/inbox?recursive=true&include=.*txt$")...
```

AUTOMATIC TYPE CONVERTERS

Camel has a built-in type-converter mechanism that ships with more than 350 converters. You no longer need to configure type-converter rules to go from byte arrays to strings, for example. And if you need to convert to types that Camel doesn't support, you can create your own type converter. The best part is that it works under the hood, so you don't have to worry about it.

The Camel components also use this feature; they can accept data in most types and convert the data to a type they're capable of using. This feature is one of the top favorites in the Camel community. You may even start wondering why it wasn't provided in Java itself! Chapter 3 covers more about type converters.

LIGHTWEIGHT CORE IDEAL FOR MICROSERVICES

Camel's core can be considered lightweight, with the total library coming in at about 4.9 MB and having only 1.3 MB of runtime dependencies. This makes Camel easy to embed or deploy anywhere you like, such as in a standalone application, microservice, web application, Spring application, Java EE application, OSGi, Spring Boot, WildFly, and in cloud platforms such as AWS, Kubernetes, and Cloud Foundry. Camel was designed not to be a server or ESB but instead to be embedded in whatever runtime you choose. You just need Java.

CLOUD READY

In addition to Camel being cloud-native (covered in chapter 18), Camel also provides many components for connecting with SaaS providers. For example, with Camel you can hook into the following:

- Amazon DynamoDB, EC2, Kinesis, SimpleDB, SES, SNS, SQS, SWF, and S3
- Braintree (PayPal, Apple, Android Pay, and so on)
- Dropbox
- Facebook
- GitHub
- Google Big Query, Calendar, Drive, Mail, and Pub Sub
- HipChat
- LinkedIn
- Salesforce
- Twitter
- And more...

TEST KIT

Camel provides a test kit that makes it easier for you to test your

own Camel applications. The same test kit is used extensively to test Camel itself, and it includes more than 18,000 unit tests. The test kit contains test-specific components that, for example, can help you mock real endpoints. It also allows you to set up expectations that Camel can use to determine whether an application satisfied the requirements or failed. Chapter 9 covers testing with Camel.

VIBRANT COMMUNITY

Camel has an active community. It's a long-lived one too. It has been active (and growing) for more than 10 years at the time of writing. Having a strong community is essential if you intend to use any open source project in your application. Inactive projects have little community support, so if you run into issues, you're on your own. With Camel, if you're having any trouble, users and developers alike will come to your aid. For more information on Camel's community, see appendix B.

Now that you've seen the main features that make up Camel, you'll get more hands-on by looking at the Camel distribution and trying an example.

1.2 *Getting started*

This section shows you how to get your hands on a Camel distribution and explains what's inside. Then you'll run an example using Apache Maven. After this, you'll know how to run any of the examples from the book's source code.

Let's first get the Camel distribution.

1.2.1 **GETTING CAMEL**

Camel is available from the official Apache Camel website at <http://camel.apache.org/download.html>. On that page, you'll see a list of all the Camel releases and the downloads for the latest release.

For the purposes of this book, we'll be using Camel 2.20.1. To

get this version, click the Camel 2.20.1 Release link. Near the bottom of the page, you'll find two binary distributions: the zip distribution is for Windows users, and the tar.gz distribution is for macOS/Linux users. After you've downloaded one of the distributions, extract it to a location on your hard drive.

Open a command prompt and go to the location where you extracted the Camel distribution. Issuing a directory listing here will give you something like this:

```
[janstey@ghost apache-camel-2.20.1]$ ls  
doc examples lib LICENSE.txt NOTICE.txt README.txt
```

As you can see, the distribution is small, and you can probably guess what each directory contains already. Here are the details:

- *doc*—Contains the Camel manual in HTML format. This manual is a download of a large portion of the Apache Camel website at the time of release. As such, it's a decent reference for those unable to access the Camel website (or if you misplaced your copy of *Camel in Action*).
- *examples*—Includes 97 Camel examples.
- *lib*—Contains all Camel libraries. You'll see later in the chapter how Maven can be used to easily download dependencies for the components outside the core.
- *LICENSE.txt*—Contains the license of the Camel distribution. Because this is an Apache project, the license is the Apache License, version 2.0.
- *NOTICE.txt*—Contains copyright information about the third-party dependencies included in the Camel distribution.
- *README.txt*—Contains a short intro to Camel and a list of helpful links to get new users up and running.

Now let's try the first Camel example from this book.

1.2.2 YOUR FIRST CAMEL RIDE

So far, we've shown you how to get a Camel distribution and

offered a peek at what's inside. At this point, feel free to explore the distribution; all examples have instructions to help you figure them out.

From this point on, though, we won't be using the distribution at all. All the examples in the book's source use Apache Maven, which means that Camel libraries will be downloaded automatically for you—there's no need to make sure the Camel distribution's libraries are on the classpath.

You can get the book's source code from the GitHub project that's hosting the source (<https://github.com/camelinaction/camelinaction2>).

The first example you'll look at can be considered the “hello world” of integrations: routing files. Suppose you need to read files from one directory (data/inbox), process them in some way, and write the result to another directory (data/outbox). For simplicity, you'll skip the processing, so your output will be merely a copy of the original file. [Figure 1.4](#) illustrates this process.

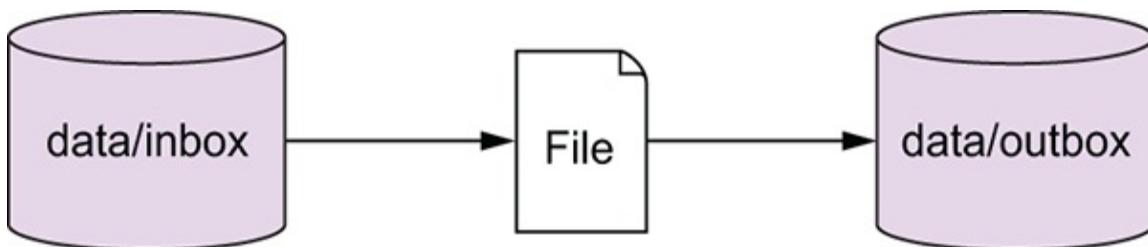


Figure 1.4 Files are routed from the data/inbox directory to the data/outbox directory.

It looks simple, right? Here's a possible solution using pure Java (with no Camel).

[Listing 1.1](#) Routing files from one folder to another in plain Java

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
```

```

import java.io.OutputStream;

public class FileCopier {

    public static void main(String args[]) throws Exception
    {
        File inboxDirectory = new File("data/inbox");
        File outboxDirectory = new File("data/outbox");
        outboxDirectory.mkdir();
        File[] files = inboxDirectory.listFiles();
        for (File source : files) {
            if (source.isFile()) {
                File dest = new File(
                    outboxDirectory.getPath()
                    + File.separator
                    + source.getName());
                copyFile(source, dest);
            }
        }
    }

    private static void copyFile(File source, File dest)
        throws IOException {
        OutputStream out = new FileOutputStream(dest);
        byte[] buffer = new byte[(int) source.length()];
        FileInputStream in = new FileInputStream(source);
        in.read(buffer);
        try {
            out.write(buffer);
        } finally {
            out.close();
            in.close();
        }
    }
}

```

This `FileCopier` example is a simple use case, but it still results in 37 lines of code. You have to use low-level file APIs and ensure that resources get closed properly—a task that can easily go wrong. Also, if you want to poll the data/inbox directory for new files, you need to set up a timer and keep track of which files you've already copied. This simple example is getting more complex.

Integration tasks like these have been done thousands of times before; you shouldn't ever need to code something like this by

hand. Let's not reinvent the wheel here. Let's see what a polling solution looks like if you use an integration framework such as Apache Camel.

Listing 1.2 Routing files from one folder to another with Apache Camel

```
import org.apache.camel.CamelContext;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.impl.DefaultCamelContext;

public class FileCopierWithCamel {

    public static void main(String args[]) throws Exception
    {
        CamelContext context = new DefaultCamelContext();
        context.addRoutes(new RouteBuilder() {
            public void configure() {
                from("file:data/inbox?noop=true") 1
                    .to("file:data/outbox");
            }
        });
        context.start();
        Thread.sleep(10000);
        context.stop();
    }
}
```

1

Routes files from inbox to outbox

```
        .to("file:data/outbox");
    }
});
context.start();
Thread.sleep(10000);
context.stop();
}
}
```

Most of this code is boilerplate stuff when using Camel. Every Camel application uses a `CamelContext` that's subsequently started and then stopped. You also add a `sleep` method to allow your simple Camel application time to copy the files. What you should focus on in [listing 1.2](#) is the *route 1*.

Routes in Camel are defined in such a way that they flow when read. This route can be read like this: consume messages from file location `data/inbox` with the `noop` option set, and send to file location `data/outbox`. The `noop` option tells Camel to leave

the source file as is. If you didn't use this option, the file would be moved. Most people who've never seen Camel before will be able to understand what this route does. You may also want to note that, excluding the boilerplate code, you created a file-polling route in just two lines of Java code ①.

To run this example, you need to download and install Apache Maven from the Maven site at <http://maven.apache.org/download.html>. When you have Maven up and working, open a terminal and browse to the chapter1/file-copy directory of the book's source. If you take a directory listing here, you'll see several things:

- *data*—Contains the inbox directory, which itself contains a single file named message1.xml.
- *src*—Contains the source code for the listings shown in this chapter.
- *pom.xml*—Contains information necessary to build the examples. This is the Maven Project Object Model (POM) XML file.

NOTE We used Maven 3.5.0 during the development of the book. Different versions of Maven may not work or appear exactly as we've shown.

The POM is shown in the following listing.

Listing 1.3 The Maven POM required to use Camel's core library

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

```
<groupId>com.camelinaction</groupId>
```

1

Parent POM

```
<artifactId>chapter1</artifactId>
<version>2.0.0</version>
</parent>

<artifactId>chapter1-file-copy</artifactId>
<name>Camel in Action 2 :: Chapter 1 :: File Copy Example</name>

<dependencies>
  <dependency>
    <groupId>org.apache.camel</groupId>
```

2

Camel's core library

```
<artifactId>camel-core</artifactId>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
```

2

3

Logging support

```
<artifactId>slf4j-log4j12</artifactId>
</dependency>
</dependencies>
</project>
```

3

Maven itself is a complex topic, and we don't go into great detail here. We'll give you enough information to be productive with the examples in this book. Chapter 8 also covers using Maven to develop Camel applications, so there's a good deal of information there too.

The Maven POM in [listing 1.3](#) is probably one of the shortest POMs you'll ever see—almost everything uses the defaults provided by Maven. Besides those defaults, some settings are configured in the parent POM 1. Probably the most important

section to point out here is the dependency on the Camel library **②**. This dependency element tells Maven to do the following:

1. Create a search path based on the `groupId`, `artifactId`, and `version`. The `version` element is set to the `camel-version` property, which is defined in the POM referenced in the parent element **①**, and resolves to `2.20.1`. The type of dependency isn't specified, so the JAR file type is assumed. The search path is `org/apache/camel/camel-core/2.20.1/camel-core-2.20.1.jar`.
2. Because [listing 1.3](#) defines no special places for Maven to look for the Camel dependencies, it looks in Maven's central repository, located at <http://repo1.maven.org/maven2>.
3. Combining the search path and the repository URL, Maven tries to download <http://repo1.maven.org/maven2/org/apache/camel/camel-core/2.20.1/camel-core-2.20.1.jar>.
4. This JAR is saved to Maven's local download cache, which is typically located in the home directory under the `.m2/repository`. This is `~/.m2/repository` on Linux/macOS, and `C:\Users\<Username>\.m2\repository` on recent versions of Windows.
5. When the application code in [listing 1.2](#) is started, the Camel JAR is added to the classpath.

To run the example in [listing 1.2](#), change to the `chapter1/file-copy` directory and use the following command:

```
mvn compile exec:java
```

This instructs Maven to compile the source in the `src` directory and to execute the `FileCopierWithCamel` class with the camel-core JAR on the classpath.

NOTE To run any of the examples in this book, you need an internet connection. Apache Maven will download many JAR dependencies of the examples. The whole set of examples will download several hundred megabytes of libraries.

Run the Maven command from the chapter1/file-copy directory, and after it completes, browse to the data/outbox folder to see the file copy that's just been made. Congratulations—you've run your first Camel example! It's a simple one, but knowing how it's set up will enable you to run pretty much any of the book's examples.

We now need to cover Camel basics and the integration space in general to ensure that you're well prepared for using Camel. We'll turn our attention to the message model, the architecture, and a few other Camel concepts. Most of the abstractions are based on known EIP concepts and retain their names and semantics. We'll start with Camel's message model.

1.3 Camel's message model

Camel uses two abstractions for modeling messages, both of which we cover in this section:

- `org.apache.camel.Message`—The fundamental entity containing the data being carried and routed in Camel.
- `org.apache.camel.Exchange`—The Camel abstraction for an exchange of messages. This exchange of messages has an *in* message, and as a reply, an *out* message.

We'll start by looking at messages so you can understand the way data is modeled and carried in Camel. Then we'll show you how a “conversation” is modeled in Camel by the exchange.

1.3.1 MESSAGE

Messages are the entities used by systems to communicate with each other when using messaging channels. Messages flow in one direction, from a sender to a receiver, as illustrated in [figure 1.5](#).

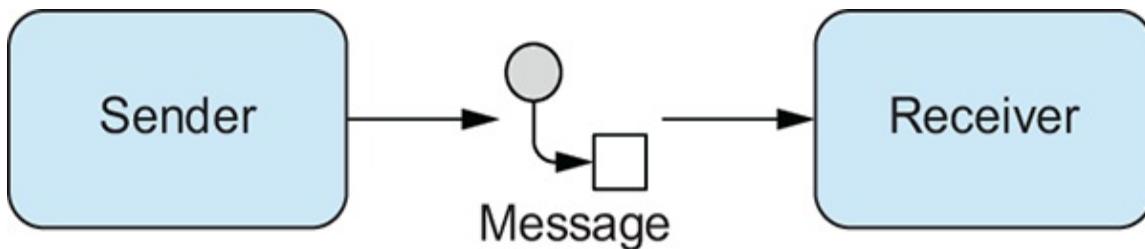


Figure 1.5 Messages are entities used to send data from one system to another.

Messages have a body (a payload), headers, and optional attachments, as illustrated in figure 1.6.

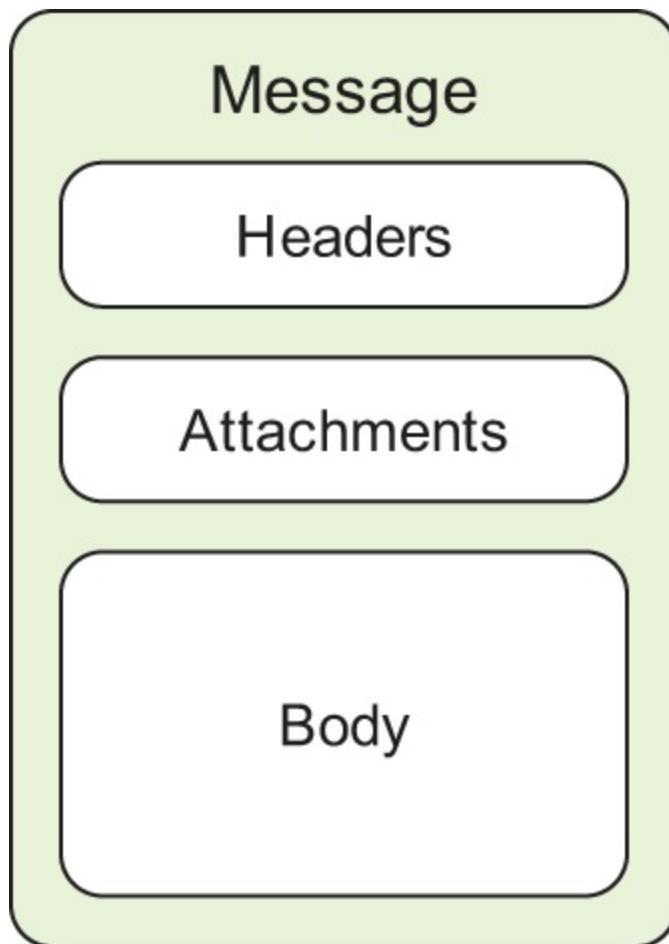


Figure 1.6 A message can contain headers, attachments, and a body.

Messages are uniquely identified with an identifier of type `java.lang.String`. The identifier's uniqueness is enforced and guaranteed by the message creator, it's protocol dependent, and it doesn't have a guaranteed format. For protocols that don't define a unique message identification scheme, Camel uses its

own ID generator.

HEADERS AND ATTACHMENTS

Headers are values associated with the message, such as sender identifiers, hints about content encoding, authentication information, and so on. Headers are name-value pairs; the name is a unique, case-insensitive string, and the value is of type `java.lang.Object`. Camel imposes no constraints on the type of the headers. There are also no constraints on the size of headers or on the number of headers included with a message. Headers are stored as a map within the message. A message can also have optional attachments, which are typically used for the web service and email components.

BODY

The body is of type `java.lang.Object`, so a message can store any kind of content and any size. It's up to the application designer to make sure that the receiver can understand the content of the message. When the sender and receiver use different body formats, Camel provides mechanisms to transform the data into an acceptable format, and in those cases the conversion happens automatically with type converters, behind the scenes. Chapter 3 fully covers message transformation.

FAULT FLAG

Messages also have a fault flag. A few protocols and specifications, such as SOAP Web Services, distinguish between *output* and *fault* messages. They're both valid responses to invoking an operation, but the latter indicates an unsuccessful outcome. In general, faults aren't handled by the integration infrastructure. They're part of the contract between the client and the server and are handled at the application level.

During routing, messages are contained in an exchange.

1.3.2 EXCHANGE

An *exchange* in Camel is the message's container during routing. An exchange also provides support for the various types of interactions between systems, also known as *message exchange patterns (MEPs)*. MEPs are used to differentiate between one-way and request-response messaging styles. The Camel exchange holds a pattern property that can be either of the following:

- `InOnly`—A one-way message (also known as an event message). For example, JMS messaging is often one-way messaging.
- `InOut`—A request-response message. For example, HTTP-based transports are often request-reply: a client submits a web request, waiting for the reply from the server.

Figure 1.7 illustrates the contents of an exchange in Camel.

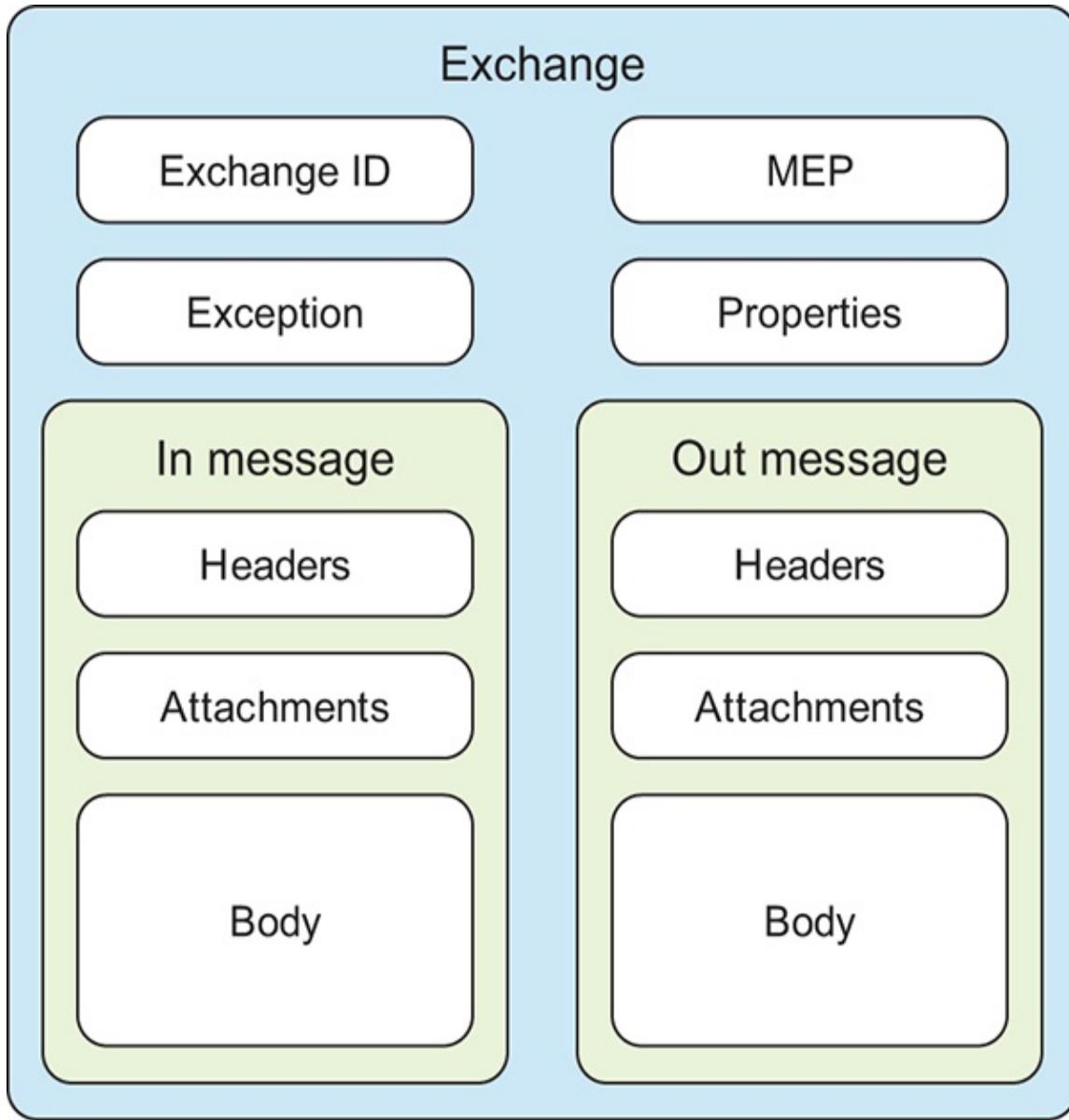


Figure 1.7 A Camel exchange has an **ID**, **MEP**, exception, and properties. It also has an *in* message to store the incoming message, and an *out* message to store the reply.

Let's look at the elements of [figure 1.7](#) in more detail:

- **Exchange ID**—A unique ID that identifies the exchange. Camel automatically generates the unique ID.
- **MEP**—A pattern that denotes whether you're using the `InOnly` or `InOut` messaging style. When the pattern is `InOnly`, the exchange contains an *in* message. For `InOut`, an *out* message also exists that contains the reply message for the caller.

- *Exception*—If an error occurs at any time during routing, an Exception will be set in the exception field.
- *Properties*—Similar to message headers, but they last for the duration of the entire exchange. Properties are used to contain global-level information, whereas message headers are specific to a particular message. Camel itself adds various properties to the exchange during routing. You, as a developer, can store and retrieve properties at any point during the lifetime of an exchange.
- *In message*—This is the input message, which is mandatory. The in message contains the request message.
- *Out message*—This is an optional message that exists only if the MEP is `InOut`. The out message contains the reply message.

The exchange is the same for the entire lifecycle of routing, but the messages can change, for instance, if messages are transformed from one format to another.

We discussed Camel’s message model before the architecture because we want you to have a solid understanding of what a message is in Camel. After all, the most important aspect of Camel is routing messages. You’re now well prepared to learn more about Camel and its architecture.

1.4 Camel’s architecture

You’ll first take a look at the high-level architecture and then drill down into the specific concepts. After you’ve read this section, you should be caught up on the integration lingo and be ready for chapter 2, where you’ll explore Camel’s routing capabilities.

1.4.1 ARCHITECTURE FROM 10,000 FEET

We think that architectures are best viewed first from high above. [Figure 1.8](#) shows a high-level view of the main concepts that make up Camel’s architecture.

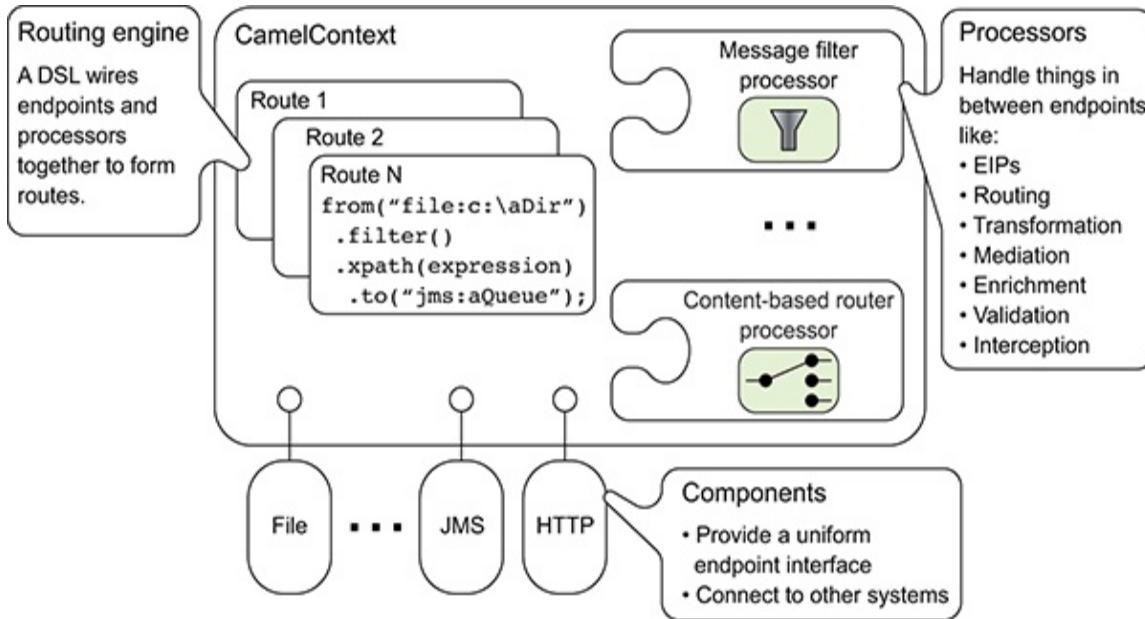


Figure 1.8 At a high level, Camel is composed of routes, processors, and components. All of these are contained within `camelContext`.

The routing engine uses routes as specifications indicating where messages are routed. Routes are defined using one of Camel's DSLs. Processors are used to transform and manipulate messages during routing as well as to implement all the EIPs, which have corresponding names in the DSLs. Components are the extension points in Camel for adding connectivity to other systems. To expose these systems to the rest of Camel, components provide an endpoint interface.

With that high-level view out of the way, let's take a closer look at the individual concepts in figure 1.8.

1.4.2 CAMEL CONCEPTS

Figure 1.8 reveals many new concepts, so let's take some time to go over them one by one. We'll start with `camelContext`, which is Camel's runtime system.

CAMEL CONTEXT

You may have guessed that `camelContext` is a container of sorts, judging from figure 1.8. You can think of it as Camel's runtime system, which keeps all the pieces together.

Figure 1.9 shows the most notable services that `camelContext` keeps together.

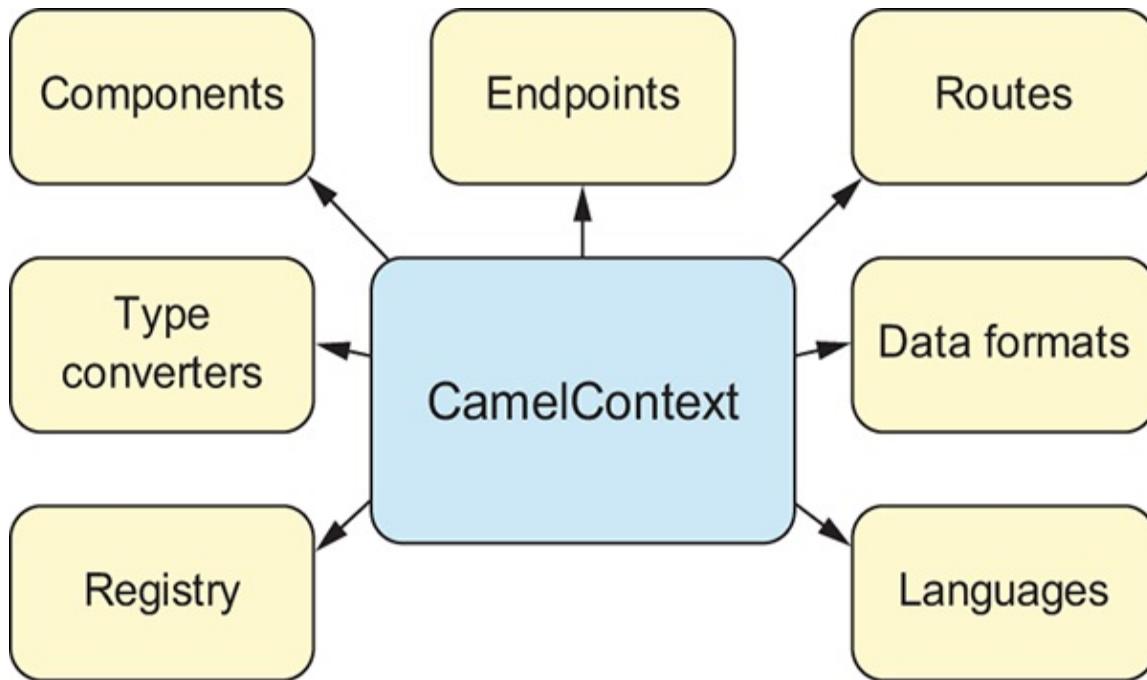


Figure 1.9 `camelContext` provides access to many useful services, the most notable being components, type converters, a registry, endpoints, routes, data formats, and languages.

As you can see, `CamelContext` has a lot of services to keep track of. These are described in table 1.1.

Table 1.1 The services that `camelContext` provides

Service	Description
Components	Contains the components used. Camel is capable of loading components on the fly either by autodiscovery on the classpath or when a new bundle is activated in an OSGi container. Chapter 6 covers components in more detail.
Endpoints	Contains the endpoints that have been used.

Routes	Contains the routes that have been added. Chapter 2 covers routes.
Type converters	Contains the loaded type converters. Camel has a mechanism that allows you to manually or automatically convert from one type to another. Type converters are covered in chapter 3.
Data formats	Contains the loaded data formats. Data formats are covered in chapter 3.
Registry	Contains a registry that allows you to look up beans. We cover registries in chapter 4.
Languages	Contains the loaded languages. Camel allows you to use many languages to create expressions. You'll get a glimpse of the XPath language in just a moment. A complete reference to Camel's own Simple expression language is available in appendix A.

The details of each service are discussed throughout the book. Let's now take a look at routes and Camel's routing engine.

ROUTING ENGINE

Camel's routing engine is what moves messages under the hood. This engine isn't exposed to the developer, but you should be aware that it's there and that it does all the heavy lifting, ensuring that messages are routed properly.

ROUTES

Routes are obviously a core abstraction for Camel. The simplest way to define a *route* is as a chain of processors. There are many reasons for using routers in messaging applications. By decoupling clients from servers, and producers from consumers, routes can do the following:

- Decide dynamically what server a client will invoke
- Provide a flexible way to add extra processing
- Allow for clients and servers to be developed independently
- Foster better design practices by connecting disparate systems that do one thing well
- Enhance features and functionality of some systems (such as message brokers and ESBs)
- Allow for clients of servers to be stubbed out (using mocks) for testing purposes

Each route in Camel has a unique identifier that's used for logging, debugging, monitoring, and starting and stopping routes. Routes also have exactly one input source for messages, so they're effectively tied to an input endpoint. That said, there's some syntactic sugar for having multiple inputs to a single route. Take the following route, for example:

```
from("jms:queue:A", "jms:queue:B",
    "jms:queue:C").to("jms:queue:D");
```

Under the hood, Camel clones the route definition into three separate routes. So, it behaves similarly to three separate routes as follows:

```
from("jms:queue:A").to("jms:queue:D");
from("jms:queue:B").to("jms:queue:D");
from("jms:queue:C").to("jms:queue:D");
```

Even though it's perfectly legal in Camel 2.x, we don't recommend using multiple inputs per route. This ability will be removed in the next major version of Camel. To define these routes, we use a DSL.

DOMAIN-SPECIFIC LANGUAGE

To wire processors and endpoints together to form routes, Camel defines a DSL. The term *DSL* is used a bit loosely here. In Camel, DSL means a fluent Java API that contains methods named for EIP terms.

Consider this example:

```
from("file:data/inbox")
    .filter().xpath("/order[not(@test)]")
    .to("jms:queue:order");
```

Here, in a single Java statement, you define a route that consumes files from a file endpoint. Messages are then routed to the filter EIP, which will use an XPath predicate to test whether the message is not a test order. If a message passes the test, it's forwarded to the JMS endpoint. Messages failing the filter test are dropped.

Camel provides multiple DSL languages, so you could define the same route by using the XML DSL, like this:

```
<route>
    <from uri="file:data/inbox"/>
    <filter>
        <xpath>/order[not(@test)]</xpath>
        <to uri="jms:queue:order"/>
    </filter>
</route>
```

The DSLs provide a nice abstraction for Camel users to build applications with. Under the hood, though, a route is composed of a graph of processors. Let's take a moment to see what a processor is.

PROCESSOR

The *processor* is a core Camel concept that represents a node capable of using, creating, or modifying an incoming exchange. During routing, exchanges flow from one processor to another; as such, you can think of a route as a graph having specialized

processors as the nodes, and lines that connect the output of one processor to the input of another. Processors could be implementations of EIPs, producers for specific components, or your own custom creation. [Figure 1.10](#) shows the flow between processors.

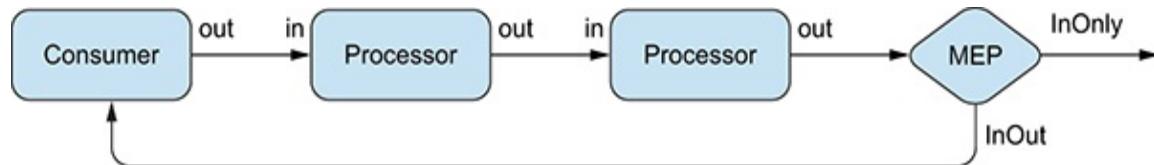


Figure 1.10 Flow of an exchange through a route. Notice that the MEP determines whether a reply will be sent back to the caller of the route.

A route first starts with a consumer (think “from” in the DSL) that populates the initial exchange. At each processor step, the out message from the previous step is the in message of the next. In many cases, processors don’t set an out message, so in this case the in message is reused. At the end of a route, the MEP of the exchange determines whether a reply needs to be sent back to the caller of the route. If the MEP is `InOnly`, no reply will be sent back. If it’s `InOut`, Camel will take the out message from the last step and return it.

NOTE Producers and consumers in Camel may seem a bit counterintuitive at first. After all, shouldn’t producers be the first node and consumers be consuming messages at the end of a route? Don’t worry—you’re not the first to think like this! Just think of these concepts from the point of view of communicating with external systems. Consumers consume messages from external systems and bring them into the route. Producers, on the other hand, send (produce) messages to external systems.

How do exchanges get in or out of this processor graph? To find out, you need to look at components and endpoints.

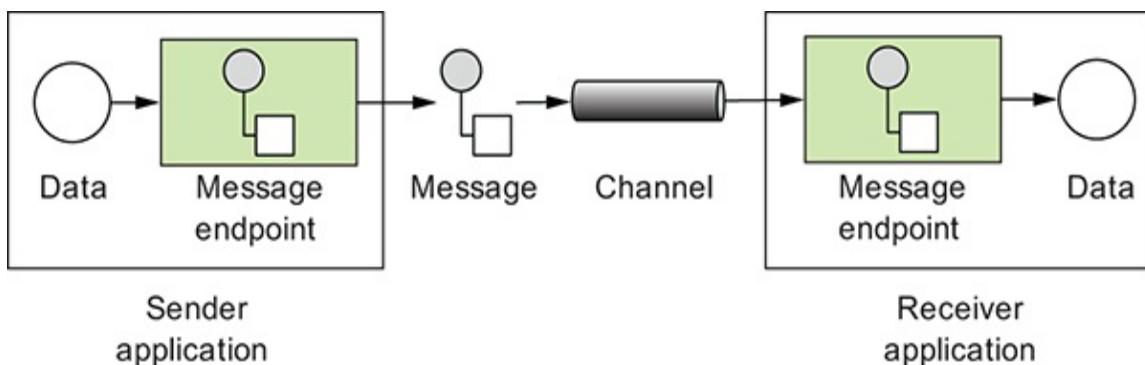
COMPONENT

Components are the main extension point in Camel. To date, the Camel ecosystem has more than 280 components that range in function from data transports, to DSLs, to data formats, and so on. You can even create your own components for Camel—we discuss this in chapter 8.

From a programming point of view, components are fairly simple: they’re associated with a name that’s used in a URI, and they act as a factory of endpoints. For example, `FileComponent` is referred to by `file` in a URI, and it creates `FileEndpoints`. The endpoint is perhaps an even more fundamental concept in Camel.

ENDPOINT

An *endpoint* is the Camel abstraction that models the end of a channel through which a system can send or receive messages. This is illustrated in [figure 1.11](#).



[Figure 1.11](#) An endpoint acts as a neutral interface allowing systems to integrate.

In Camel, you configure endpoints by using URIs, such as `file:data/inbox?delay=5000`, and you also refer to endpoints this way. At runtime, Camel looks up an endpoint based on the URI notation. [Figure 1.12](#) shows how this works.

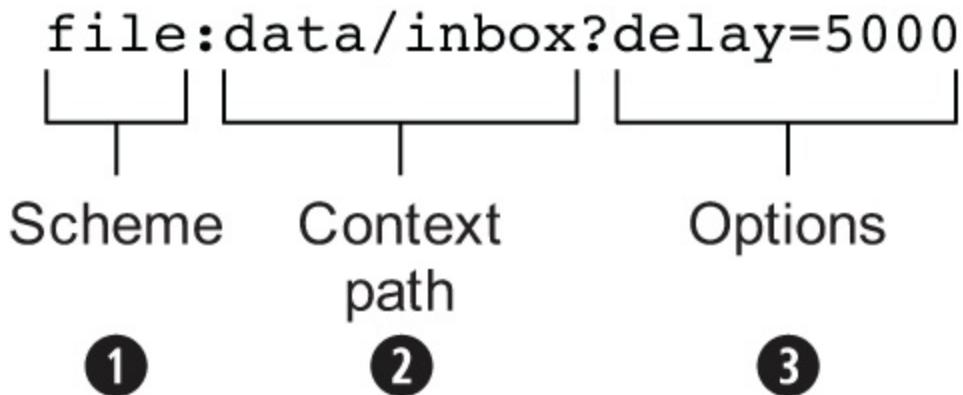


Figure 1.12 Endpoint URIs are divided into three parts: a scheme, a context path, and options.

The scheme ❶ denotes which Camel component handles that type of endpoint. In this case, the scheme of `file` selects `FileComponent`. `FileComponent` then works as a factory, creating `FileEndpoint` based on the remaining parts of the URI. The context path `data/inbox` ❷ tells `FileComponent` that the starting folder is `data/inbox`. The option, `delay=5000` ❸ indicates that files should be polled at a 5-second interval.

There's more to an endpoint than meets the eye. [Figure 1.13](#) shows how an endpoint works together with an exchange, producers, and consumers. At first glance, [figure 1.13](#) may seem a bit overwhelming, but it will all make sense in a few minutes. In a nutshell, an endpoint acts as a factory for creating consumers and producers that are capable of receiving and sending messages to a particular endpoint. We didn't mention producers or consumers in the high-level view of Camel in [figure 1.8](#), but they're important concepts. We'll go over them next.

PRODUCER

A *producer* is the Camel abstraction that refers to an entity capable of sending a message to an endpoint. [Figure 1.13](#) illustrates where the producer fits in with other Camel concepts.

When a message is sent to an endpoint, the producer handles the details of getting the message data compatible with that

particular endpoint. For example, `FileProducer` will write the message body to a file. `JmsProducer`, on the other hand, will map the Camel message to `javax.jms.Message` before sending it to a JMS destination. This is an important feature in Camel, because it hides the complexity of interacting with particular transports. All you need to do is route a message to an endpoint, and the producer does the heavy lifting.

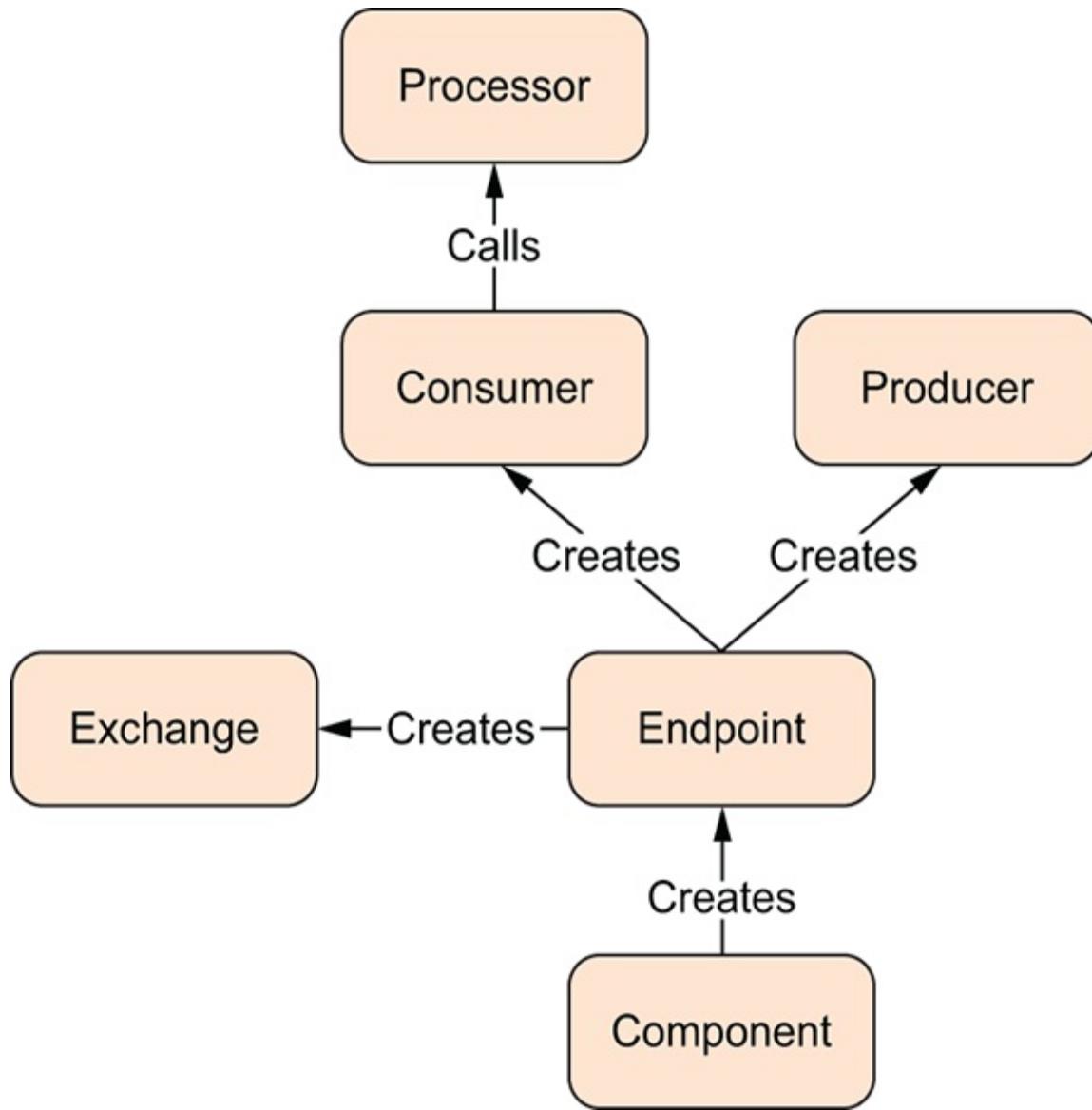


Figure 1.13 How endpoints work with producers, consumers, and an exchange

CONSUMER

A *consumer* is the service that receives messages produced by some external system, wraps them in an exchange, and sends them to be processed. Consumers are the source of the exchanges being routed in Camel.

Looking back at [figure 1.13](#), you can see where the consumer fits in with other Camel concepts. To create a new exchange, a consumer will use the endpoint that wraps the payload being consumed. A processor is then used to initiate the routing of the exchange in Camel via the routing engine.

Camel has two kinds of consumers: event-driven consumers and polling consumers. The differences between these consumers are important, because they help solve different problems.

EVENT-DRIVEN CONSUMER

The most familiar consumer is probably the *event-driven consumer*, which is illustrated in [figure 1.14](#).

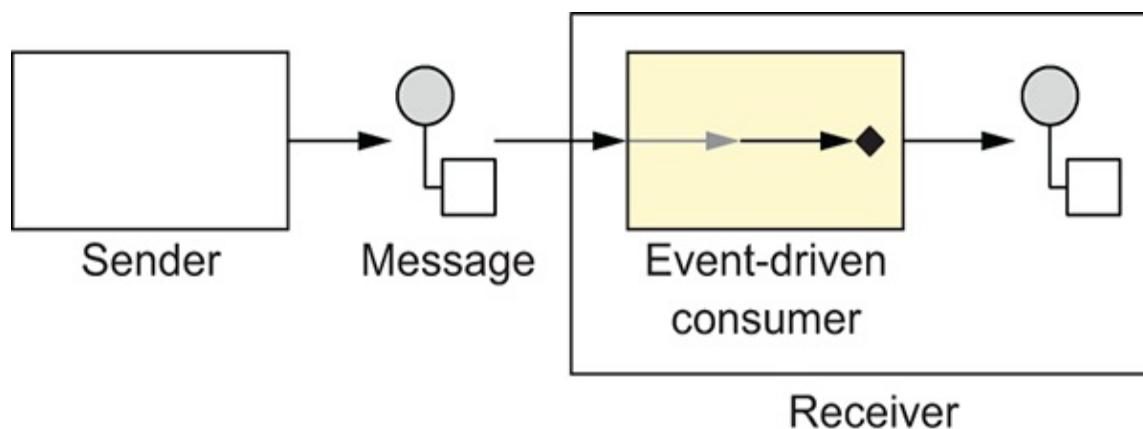


Figure 1.14 An event-driven consumer remains idle until a message arrives, at which point it wakes up and consumes the message.

This kind of consumer is mostly associated with client-server architectures and web services. It's also referred to as an *asynchronous receiver* in the EIP world. An event-driven consumer listens on a particular messaging channel, such as a TCP/IP port, JMS queue, Twitter handle, Amazon SQS queue, WebSocket, and so on. It then waits for a client to send messages

to it. When a message arrives, the consumer wakes up and takes the message for processing.

POLLING CONSUMER

The other kind of consumer is the *polling consumer*, illustrated in figure 1.15.

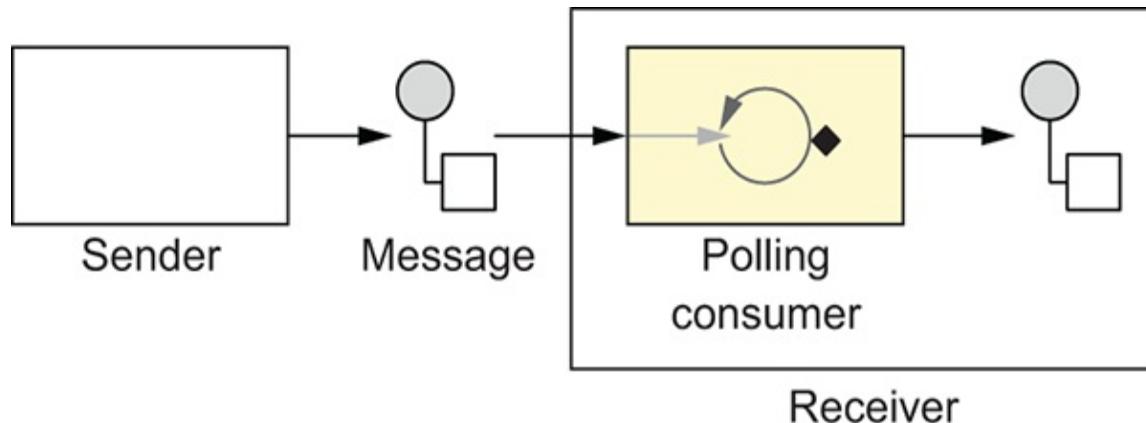


Figure 1.15 A polling consumer actively checks for new messages.

In contrast to the event-driven consumer, the polling consumer actively goes and fetches messages from a particular source, such as an FTP server. The polling consumer is also known as a *synchronous receiver* in EIP lingo, because it won't poll for more messages until it's finished processing the current message. A common flavor of the polling consumer is the scheduled polling consumer, which polls at scheduled intervals. File, FTP, and email components all use scheduled polling consumers.

We've now covered all of Camel's core concepts. With this new knowledge, you can revisit your first Camel ride and see what's happening.

1.5 Your first Camel ride, revisited

Recall that in your first Camel ride (section 1.2.2), you read files from one directory (data/inbox) and wrote the results to another directory (data/outbox). Now that you know the core Camel concepts, you can put this example in perspective.

Take another look at the Camel application in the following listing.

Listing 1.4 Routing files from one folder to another with Camel

```
import org.apache.camel.CamelContext;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.impl.DefaultCamelContext;

public class FileCopierWithCamel {

    public static void main(String args[]) throws Exception
    {
        CamelContext context = new DefaultCamelContext();
        context.addRoutes(new RouteBuilder() {
            public void configure() {
                from("file:data/inbox?noop=true") ①
①
```

1

Java DSL route

```
                .to("file:data/outbox"); ①
            }
        });
        context.start();
        Thread.sleep(10000);
        context.stop();
    }
}
```

In this example, you first create `CamelContext`, which is the Camel runtime. You then add the routing logic by using `RouteBuilder` and the Java DSL **①**. By using the DSL, you can cleanly and concisely let Camel instantiate components, endpoints, consumers, producers, and so on. All you have to focus on is defining the routes that matter for your integration projects. Under the hood, though, Camel is accessing the `FileComponent`, and using it as a factory to create the endpoint and its producer. The same `FileComponent` is used to create the consumer side as well.

NOTE You may be wondering whether you always need that ugly `Thread.sleep` call. Thankfully, the answer is no! The example was created in this way to demonstrate the low-level mechanics of Camel's API. If you were deploying your Camel route to another container or runtime (as you'll see in chapters 7 and 15) or as a unit test (covered in detail in chapter 9, but also used in chapter 2), you wouldn't need to explicitly wait a set amount of time. Even for standalone routes not deployed to any container, there's a better way. Camel provides the `org.apache.camel.main.Main` helper class to start up a route of your choosing and wait for the JVM to terminate. We cover this in chapter 7.

1.6 Summary

In this chapter, you met Camel. You saw how Camel simplifies integration by relying on enterprise integration patterns (EIPs). You also saw Camel's DSL, which aims to make Camel code self-documenting and keeps developers focused on what the glue code does, not how it does it.

We covered Camel's main features, what Camel is and isn't, and where it can be used. We showed how Camel provides abstractions and an API that work over a large range of protocols and data formats.

At this point, you should have a good understanding of what Camel does and its underlying concepts. Soon you'll be able to confidently browse Camel applications and get a good idea of what they do.

In the rest of the book, you'll explore Camel's features and learn practical solutions you can apply in everyday integration scenarios. We'll also explain what's going on under Camel's tough skin. To make sure you get the main concepts from each chapter, from now on we'll present you with best practices and key points in the summary.

In the next chapter, you'll investigate routing, which is an essential feature and a fun one to learn.

2

Routing with Camel

This chapter covers

- An overview of routing
- Introducing the Rider Auto Parts scenario
- The basics of FTP and JMS endpoints
- Creating routes using the Java DSL
- Configuring routes in XML
- Routing using EIPs

One of the most important features of Camel is routing; without it, Camel would be a library of transport connectors. In this chapter, you'll dive into routing with Camel.

Routing happens in many aspects of everyday life. When you mail a letter, for instance, it may be routed through several cities before reaching its final address. An email you send will be routed through many computer network systems before reaching its final destination. In all cases, the router's function is to selectively move the message forward.

In the context of enterprise messaging systems, routing is the process by which a message is taken from an input queue and, based on a set of conditions, sent to one of several output queues, as shown in [figure 2.1](#). The input and output queues are

unaware of the conditions in between them. The conditional logic is decoupled from the message consumer and producer.

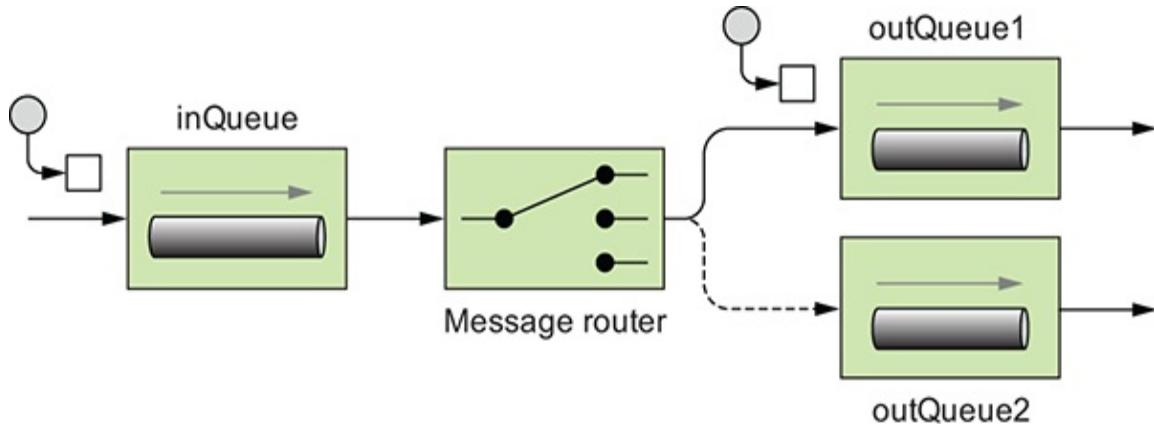


Figure 2.1 A message router consumes messages from an input channel and, depending on a set of conditions, sends the message to one of a set of output channels.

In Camel, routing is a more general concept. It's defined as a step-by-step movement of the message, which originates from an endpoint in the role of a consumer. The consumer could be receiving the message from an external service, polling for the message on a system, or even creating the message itself. This message then flows through a processing node, which could be an enterprise integration pattern (EIP), a processor, an interceptor, or another custom creation. The message is finally sent to a target endpoint that's in the role of a producer. A route may have many processing components that modify the message or send it to another location, or it may have none, in which case it would be a simple pipeline.

This chapter introduces the fictional company that we use as the running example throughout the book. To support this company's use case, you'll learn how to communicate over FTP and Java Message Service (JMS) by using Camel's endpoints. Following this, you'll look in depth at the Java-based domain-specific language (DSL) and the XML-based DSL for creating routes. We'll also give you a glimpse of how to design and implement solutions to enterprise integration problems by using EIPs and Camel. By the end of the chapter, you'll be proficient

enough to create useful routing applications with Camel.

To start, let's look at the example company used to demonstrate the concepts throughout the book.

2.1 Introducing Rider Auto Parts

Our fictional motorcycle parts business, Rider Auto Parts, supplies parts to motorcycle manufacturers. Over the years, Rider Auto Parts has changed the way it receives orders several times. Initially, orders were placed by uploading comma-separated values (CSV) files to an FTP server. The message format was later changed to XML. Currently, the company provides a website through which orders are submitted as XML messages over HTTP.

Rider Auto Parts asks new customers to use the web interface to place orders, but because of service-level agreements (SLAs) with existing customers, the company must keep all the old message formats and interfaces up and running. All of these messages are converted to an internal Plain Old Java Object (POJO) format before processing. A high-level view of the order-processing system is shown in [figure 2.2](#).

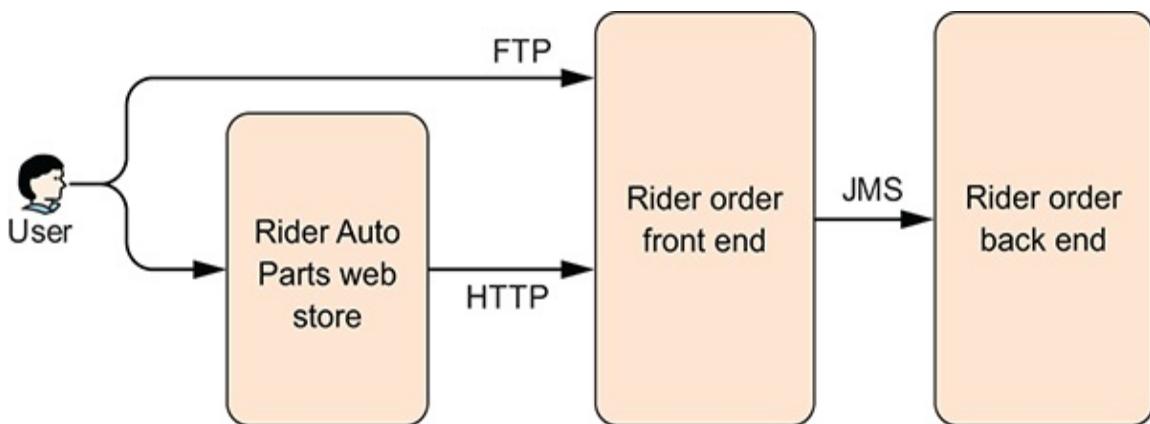


Figure 2.2 A customer has two ways of submitting orders to the Rider Auto Parts order-handling system: either by uploading the raw order file to an FTP server or by submitting an order through the Rider Auto Parts web store. All orders are eventually sent via JMS for processing at Rider Auto Parts.

Rider Auto Parts faces a common problem: over years of

operation, it has acquired software baggage in the form of transports and data formats that were popular at the time. This is no problem for an integration framework like Camel, though. In this chapter, and throughout the book, you'll help Rider Auto Parts implement its current requirements and new functionality by using Camel.

As a first assignment, you'll need to implement the FTP module in the Rider order front-end system. Later in the chapter, you'll see how back-end services are implemented too. Implementing the FTP module involves the following steps:

1. Polling the FTP server and downloading new orders
2. Converting the order files to JMS messages
3. Sending the messages to the JMS `incomingOrders` queue

To complete steps 1 and 3, you need to understand how to communicate over FTP and JMS by using Camel's endpoints. To complete the entire assignment, you need to understand routing with the Java DSL. Let's first take a look at how to use Camel's endpoints.

2.2 Understanding endpoints

As you read in chapter 1, an *endpoint* is an abstraction that models the end of a message channel through which a system can send or receive messages. This section explains how to use URIs to configure Camel to communicate over FTP and JMS. Let's first look at FTP.

2.2.1 CONSUMING FROM AN FTP ENDPOINT

One of the things that makes Camel easy to use is the endpoint URI. With an endpoint URI, you can identify the component you want to use and the way that component is configured. You can then decide to either send messages to the component configured by this URI, or to consume messages from it.

Take your first Rider Auto Parts assignment, for example. To

download new orders from the FTP server, you need to do the following:

1. Connect to the rider.com FTP server on the default FTP port of 21
2. Provide a username of `rider` and password of `secret`
3. Change the directory to `orders`
4. Download any new order files

As shown in [figure 2.3](#), you can easily configure Camel to do this by using URI notation.

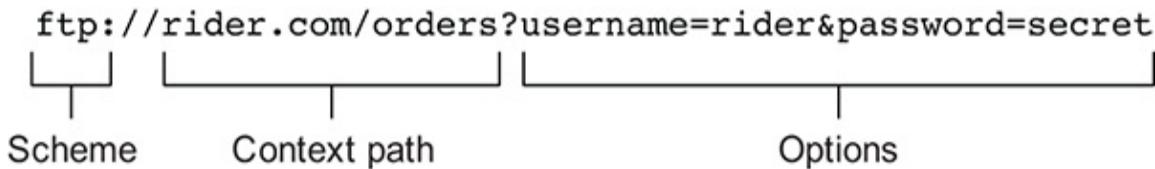


Figure 2.3 A Camel endpoint URI consists of three parts: a scheme, a context path, and a list of options.

Camel first looks up the `ftp` scheme in the component registry, which resolves to `FtpComponent`. `FtpComponent` then works as a factory, creating `FtpEndpoint` based on the remaining context path and options.

The context path of `rider.com/orders` tells `FtpComponent` that it should log into the FTP server at `rider.com` on the default FTP port and change the directory to `orders`. Finally, the only options specified are `username` and `password`, which are used to log in to the FTP server.

TIP For the FTP component, you can also specify the username and password in the context path of the URI, so the following URI is equivalent to the one in [figure 2.3](#):
`ftp://rider:secret@rider.com/orders`. Speaking of passwords, defining them in plain text isn't usually a good idea! You'll find out how to use encrypted passwords in chapter 14.

FtpComponent isn't part of the camel-core module, so you have to add a dependency to your project. Using Maven, you add the following dependency to the POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>2.20.1</version>
</dependency>
```

Although this endpoint URI would work equally well in a consumer or producer scenario, you'll be using it to download orders from the FTP server. To do so, you need to use it in a `from` node of Camel's DSL:

```
from("ftp://rider.com/orders?
username=rider&password=secret")
```

That's all you need to do to consume files from an FTP server.

The next thing you need to do, as you may recall from [figure 2.2](#), is send the orders you downloaded from the FTP server to a JMS queue. This process requires a little more setup, but it's still easy.

2.2.2 SENDING TO A JMS ENDPOINT

Camel provides extensive support for connecting to JMS-enabled providers, and we cover all the details in chapter 6. For now, though, we're going to cover just enough so that you can complete your first task for Rider Auto Parts. Recall that you need to download orders from an FTP server and send them to a JMS queue.

WHAT IS JMS?

Java Message Service (JMS) is a Java API that allows you to create, send, receive, and read messages. It also mandates that messaging is asynchronous and has specific elements of reliability, such as guaranteed and once-and-only-once delivery. JMS is probably the most widely deployed messaging solution in

the Java community.

In JMS, message consumers and producers talk to one another through an intermediary—a JMS destination. As shown in [figure 2.4](#), a destination can be either a queue or a topic. *Queues* are strictly point-to-point; each message has only one consumer. *Topics* operate on a publish/subscribe scheme; a single message may be delivered to many consumers if they've subscribed to the topic.

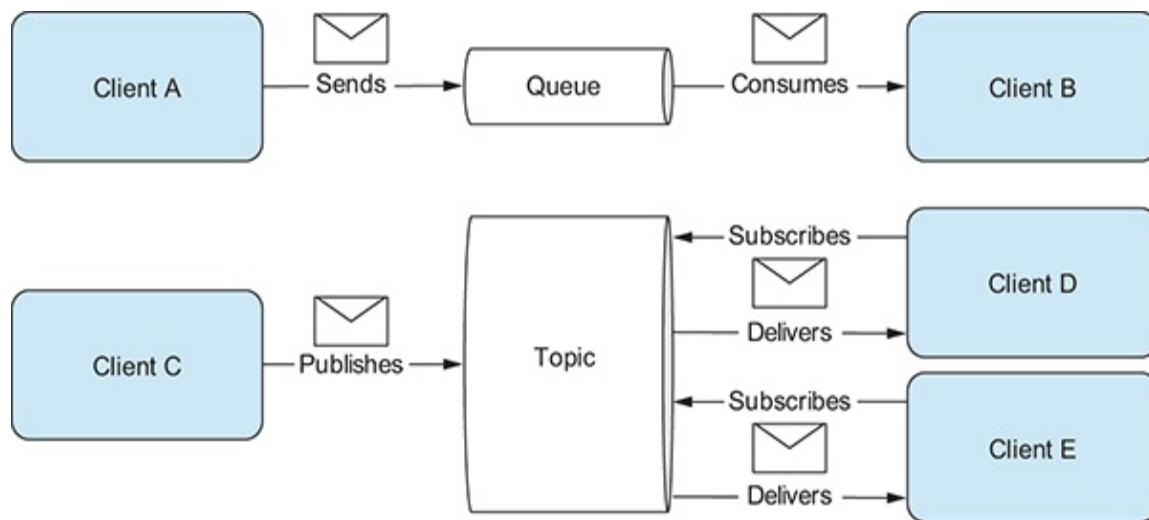


Figure 2.4 There are two types of JMS destinations: queues and topics. The *queue* is a point-to-point channel; each message has only one recipient. A *topic* delivers a copy of the message to all clients that have subscribed to receive it.

JMS also provides a `ConnectionFactory` that clients (for example, Camel) can use to create a connection with a JMS provider. JMS providers are usually referred to as *brokers* because they manage the communication between a message producer and a message consumer.

HOW TO CONFIGURE CAMEL TO USE A JMS PROVIDER

To connect Camel to a specific JMS provider, you need to configure Camel's JMS component with an appropriate `ConnectionFactory`.

Apache ActiveMQ is one of the most popular open source JMS providers, and it's the primary JMS broker that the Camel team

uses to test the JMS component. As such, we'll be using it to demonstrate JMS concepts within the book. For more information on Apache ActiveMQ, we recommend *ActiveMQ in Action* by Bruce Snyder et al. (Manning, 2011).

In the case of Apache ActiveMQ, you can create an `ActiveMQConnectionFactory` that points to the location of the running ActiveMQ broker:

```
ConnectionFactory connectionFactory =  
    new ActiveMQConnectionFactory("vm://localhost");
```

The `vm://localhost` URI means that you should connect to an embedded broker named *localhost* running inside the current JVM. The `vm` transport connector in ActiveMQ creates a broker on demand if one isn't running already, so it's handy for quickly testing JMS applications; for production scenarios, it's recommended that you connect to a broker that's already running. Furthermore, in production scenarios, we recommend that connection pooling be used when connecting to a JMS broker. See chapter 6 for details on these alternate configurations.

Next, when you create your `CamelContext`, you can add the JMS component as follows:

```
CamelContext context = new DefaultCamelContext();  
context.addComponent("jms",  
  
    JmsComponent.jmsComponentAutoAcknowledge(connectionFactory)  
);
```

The JMS component and the ActiveMQ-specific connection factory aren't part of the camel-core module. To use these, you need to add dependencies to your Maven-based project. For the plain JMS component, all you have to add is this:

```
<dependency>  
    <groupId>org.apache.camel</groupId>  
    <artifactId>camel-jms</artifactId>  
    <version>2.20.1</version>  
</dependency>
```

The connection factory comes directly from ActiveMQ, so you need the following dependency:

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-all</artifactId>
  <version>5.15.2</version>
</dependency>
```

Now that you've configured the JMS component to connect to a JMS broker, it's time to look at how URIs can be used to specify the destination.

USING URIs TO SPECIFY THE DESTINATION

After the JMS component is configured, you can start sending and receiving JMS messages at your leisure. Because you're using URIs, this is a real breeze to configure.

Let's say you want to send a JMS message to the queue named `incomingOrders`. The URI in this case would be as follows:

```
jms:queue:incomingOrders
```

This is self-explanatory. The `jms` prefix indicates that you're using the JMS component you configured before. By specifying `queue`, the JMS component knows to send to a queue named `incomingOrders`. You could even omit the `queue` qualifier, because the default behavior is to send to a queue rather than a topic.

NOTE Some endpoints can have an intimidating list of endpoint URI properties. For instance, the JMS component has more than 80 options, many of which are used only in specific JMS scenarios. Camel always tries to provide built-in defaults that fit most cases, and you can always determine the default values by browsing to the component's page in the online Camel documentation. The JMS component is discussed here:
<http://camel.apache.org/jms.html>.

Using Camel's Java DSL, you can send a message to the `incomingOrders` queue by using the `to` keyword like this:

```
...to("jms:queue:incomingOrders")
```

This can be read as sending to the JMS queue named `incomingOrders`.

Now that you know the basics of communicating over FTP and JMS with Camel, you can get back to the routing theme of this chapter and start routing messages!

2.3 Creating routes in Java

In chapter 1, you saw that `RouteBuilder` can be used to create a route and that each `CamelContext` can contain multiple routes. It may not have been obvious, though, that `RouteBuilder` isn't the final route that `CamelContext` will use at runtime; it's a builder of one or more routes, which are then added to `CamelContext`. This is illustrated in figure 2.5.

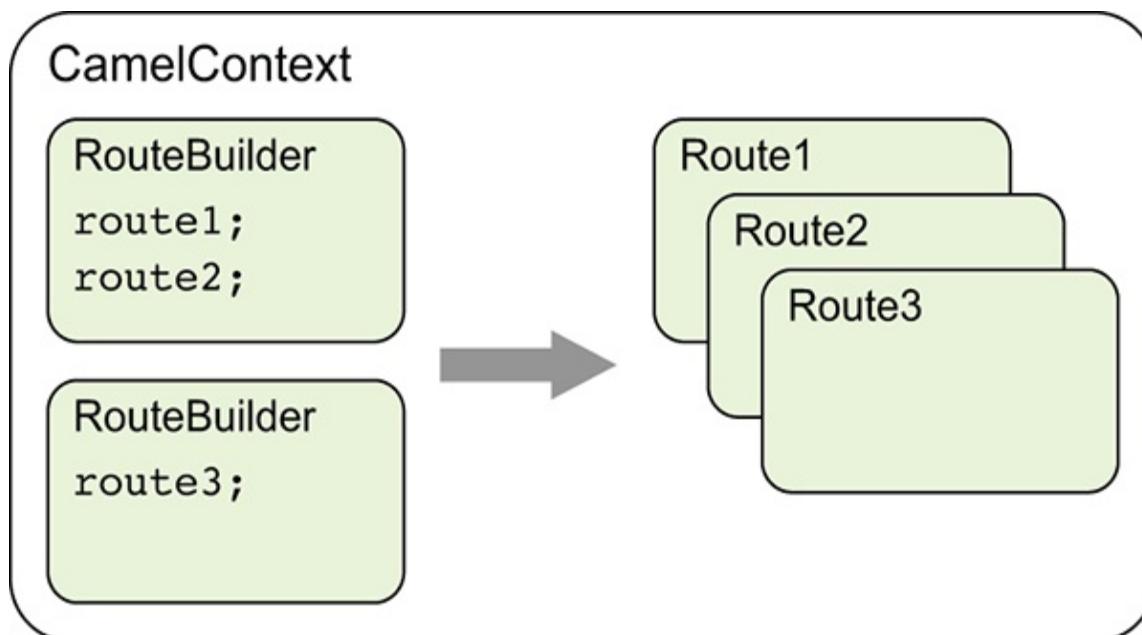


Figure 2.5 `RouteBuilders` are used to create routes in Camel. Each `RouteBuilder` can create multiple routes.

IMPORTANT This distinction between RouteBuilder and routes is an important one. The DSL code you write in RouteBuilder, whether that's with the Java or XML DSL, is merely a design-time construct that Camel uses once at startup. So, for instance, the routes that are constructed from RouteBuilder are the things that you can debug with your IDE. We cover more about debugging Camel applications in chapter 8.

The addRoutes method of CamelContext accepts RoutesBuilder, not just RouteBuilder. The RoutesBuilder interface has a single method defined:

```
void addRoutesToCamelContext(CamelContext context) throws Exception;
```

You could in theory use your own custom class to build Camel routes. Not that you'll ever want to do this, though; Camel provides the RouteBuilder class for you, which implements RoutesBuilder. The RouteBuilder class also gives you access to Camel's Java DSL for route creation.

In the next sections, you'll learn how to use RouteBuilder and the Java DSL to create simple routes. Then you'll be well prepared to take on the XML DSL in section 2.4 and routing using EIPs in section 2.6.

2.3.1 USING ROUTEBUILDER

The abstract org.apache.camel.builder.RouteBuilder class in Camel is one that you'll see frequently. You need to use it anytime you create a route in Java.

To use the RouteBuilder class, you extend a class from it and implement the configure method, like this:

```
public class MyRouteBuilder extends RouteBuilder {  
    public void configure() throws Exception {  
        ...  
    }  
}
```

You then need to add the class to `CamelContext` with the `addRoutes` method:

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new MyRouteBuilder());
```

Alternatively, you can combine the `RouteBuilder` and `CamelContext` configuration by adding an anonymous `RouteBuilder` class directly into `CamelContext`, like this:

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new RouteBuilder() {
    public void configure() throws Exception {
        ...
    }
});
```

Within the `configure` method, you define your routes by using the Java DSL. We cover the Java DSL in detail in the next section, but you can start a route now to get an idea of how it works.

In chapter 1, you should've downloaded the source code from the book's source code at GitHub and set up Apache Maven. If you didn't do this, please do so now. We'll also be using Eclipse to demonstrate Java DSL concepts.

NOTE Eclipse is a popular open source IDE that you can find at <http://eclipse.org>. During the book's development, Jon used Eclipse and Claus used IDEA. You can certainly use other Java IDEs as well, or even no IDE, but using an IDE does make Camel development a lot easier. Feel free to skip to the next section if you don't want to see the IDE-related setup. In chapter 19, you will see some additional Camel tooling you can install in Eclipse or IDEA that makes Camel development even better.

After Eclipse is set up, you should import the Maven project in the `chapter2/ftp-jms` directory of the book's source.

When the `ftp-jms` project is loaded in Eclipse, open the

src/main/java/camelinaction/RouteBuilderExample.java file. As shown in [figure 2.6](#), when you try autocomplete (Ctrl-spacebar in Eclipse) in the configure method, you'll be presented with several methods. To start a route, you should use the `from` method.

The screenshot shows a Java code editor with the following code:

```
public class RouteBuilderExample {  
    public static void main(String args[]) throws Exception {  
        CamelContext context = new DefaultCamelContext();  
  
        context.addRoutes(new RouteBuilder() {  
            @Override  
            public void configure() {  
                // try auto complete in your IDE on the line below  
                from()  
            }  
        });  
        context.start();  
    }  
}
```

The cursor is at the end of the word `from()`. A tooltip window is open, listing several `from` methods available for completion:

- from(Endpoint endpoint) : RouteDefinition - RouteBuilder
- from(Endpoint... endpoints) : RouteDefinition - RouteBuilder
- from(String uri) : RouteDefinition - RouteBuilder
- from(String... uris) : RouteDefinition - RouteBuilder
- fromF(String uri, Object... args) : RouteDefinition - RouteBuilder
- interceptFrom() : InterceptFromDefinition - RouteBuilder
- interceptFrom(String uri) : InterceptFromDefinition - RouteBuilder

Below the list, the `from` method is highlighted with a callout, and its documentation is displayed:

from
public `RouteDefinition from(String uri)`
Creates a new route from the given URI input
Parameters:
uri - the from uri
Returns:
the builder

Figure 2.6 Use autocomplete to start your route. All routes start with a `from` method.

The `from` method accepts an endpoint URI as an argument. You can add an FTP endpoint URI to connect to the Rider Auto Parts order server as follows:

```
from("ftp://rider.com/orders?  
username=rider&password=secret")
```

The `from` method returns a `RouteDefinition` object, on which you can invoke various methods that implement EIPs and other messaging concepts.

Congratulations—you're now using Camel's Java DSL! Let's take a closer look at what's going on here.

2.3.2 USING THE JAVA DSL

Domain-specific languages (DSLs) are computer languages that target a specific problem domain, rather than a general-purpose domain as most programming languages do. For example, you've probably used the regular expression DSL to match strings of text and found it to be a concise way of matching strings. Doing

the same string matching in Java wouldn't be so easy. The regular expression DSL is an *external DSL*; it has a custom syntax and so requires a separate compiler or interpreter to execute. *Internal DSLs*, in contrast, use an existing general-purpose language, such as Java, in such a way that the DSL feels like a language from a particular domain. The most obvious way of doing this is by naming methods and arguments to match concepts from the domain in question.

Another popular way of implementing internal DSLs is by using *fluent interfaces* (a.k.a. *fluent builders*). When using a fluent interface, you build up objects by chaining together method invocations. Methods of this type perform an operation and then return the current object instance.

NOTE For more information on internal DSLs, see Martin Fowler's "Domain Specific Language" entry on his bliqui (blog plus wiki) at www.martinfowler.com/bliqui/DomainSpecificLanguage.html. He also has an entry on "Fluent Interfaces" at www.martinfowler.com/bliqui/FluentInterface.html. For more information on DSLs in general, we recommend *DSLs in Action* by Debasish Ghosh (Manning, 2010).

Camel's domain is enterprise integration, so the Java DSL is a set of fluent builders that contain methods named after terms from the EIP book. In the Eclipse editor, take a look at what's available using autocomplete after a `from` method in the `RouteBuilder`. You should see something like what's shown in [figure 2.7](#). The screenshot shows a couple of EIPs—the Enricher and Recipient List—and there are many others that we'll discuss later.



Figure 2.7 After the `from` method, use your IDE's autocomplete feature to get a list of EIPs (such as Enricher and Recipient List) and other useful integration functions.

For now, select the `to` method, pass in the string `"jms:incomingOrders"`, and finish the route with a semicolon. Each Java statement that starts with a `from` method in the `RouteBuilder` creates a new route. This new route now completes your first task at Rider Auto Parts: consuming orders from an FTP server and sending them to the `incomingOrders` JMS queue. If you want, you can load up the completed example from the book's source code, in `chapter2/ftp-jms`, and open `src/main/java/camelinaction/FtpToJMSExample.java`. The code is shown in the following listing.

Listing 2.1 Polling for FTP messages and sending them to the `incomingOrders` queue

```
import javax.jms.ConnectionFactory;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.camel.CamelContext;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.jms.JmsComponent;
import org.apache.camel.impl.DefaultCamelContext;

public class FtpToJMSExample {
    public static void main(String args[]) throws Exception
    {
        CamelContext context = new DefaultCamelContext();
```

```
ConnectionFactory connectionFactory =
    new
ActiveMQConnectionFactory("vm://localhost");
    context.addComponent("jms",
JmsComponent.jmsComponentAutoAcknowledge(connectionFactory)
);

context.addRoutes(new RouteBuilder() {
    public void configure() {
        from("ftp://rider.com/orders") ①

```

①

Java statement that forms a route

```
+ "?"
username=rider&password=secret") ①
    .to("jms:incomingOrders"); ①
}
});

context.start();
Thread.sleep(10000);
context.stop();
}
}
```

NOTE Because you're consuming from `ftp://rider.com`, which doesn't exist, you can't run this example. It's useful only for demonstrating the Java DSL constructs. For runnable FTP examples, see chapter 6.

As you can see, this listing includes a bit of boilerplate setup and configuration, but the solution to the problem is concisely defined within the `configure` method as a single Java statement **①**. The `from` method tells Camel to consume messages from an FTP endpoint, and the `to` method instructs Camel to send messages to a JMS endpoint.

The flow of messages in this simple route can be viewed as a basic pipeline: the output of the consumer is fed into the

producer as input. This is depicted in figure 2.8.

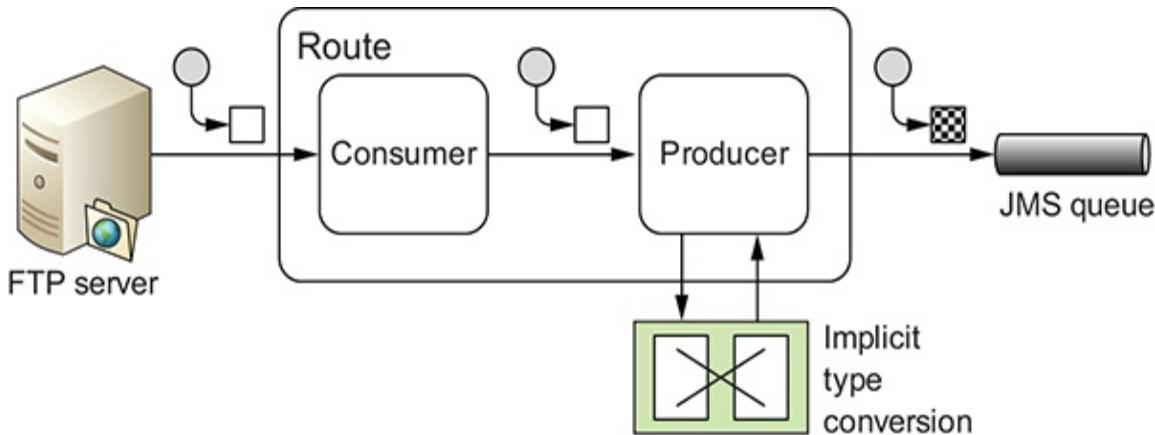


Figure 2.8 The payload conversion from file to JMS message is done automatically.

One thing you may have noticed is that we didn't do any conversion from the FTP file type to the JMS message type—this was done automatically by Camel's type-converter facility. You can force type conversions to occur at any time during a route, but often you don't have to worry about them at all. Data transformation and type conversion is covered in detail in chapter 3.

You may be thinking now that although this route is nice and simple, it'd be nice to see what's going on in the middle of the route. Fortunately, Camel always lets the developer stay in control by providing ways to hook into flows or inject behavior into features. There's a simple way of getting access to the message by using a processor, and we'll discuss that next.

ADDING A PROCESSOR

The Processor interface in Camel is an important building block of complex routes. It's a simple interface, having a single method:

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

This gives you full access to the message exchange, letting you do pretty much whatever you want with the payload or headers.

All EIPs in Camel are implemented as processors. You can even add a simple processor to your route inline, like so:

```
from("ftp://rider.com/orders?
username=rider&password=secret")
    .process(new Processor() {
        public void process(Exchange exchange) throws
Exception {
            System.out.println("We just downloaded: "
+
exchange.getIn().getHeader("CamelFileName"));
        }
    })
    .to("jms:incomingOrders");
```

This route will now print the filename of the order that was downloaded before sending it to the JMS queue.

By adding this processor into the middle of the route, you've added it to the conceptual pipeline we mentioned earlier, as illustrated in [figure 2.9](#). The output of the FTP consumer is fed into the processor as input; the processor doesn't modify the message payload or headers, so the exchange moves on to the JMS producer as input.

NOTE Many components, such as `FileComponent` and `FtpComponent`, set useful headers describing the payload on the incoming message. In the previous example, you used the `CamelFileName` header to retrieve the filename of the file that was downloaded via FTP. The component pages of the online documentation contain information about the headers set for each individual component. You'll find information about the FTP component at <http://camel.apache.org/ftp.html>.

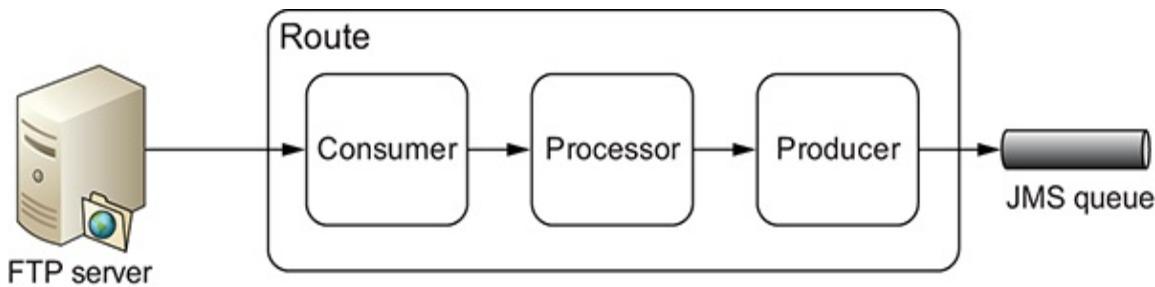


Figure 2.9 With a processor in the mix, the output of the FTP consumer is now fed into the processor, and then the output of the processor is fed into the JMS producer.

Camel's main method for creating routes is through the Java DSL. It is, after all, built into the camel-core module. There are other ways of creating routes, though, some of which may better suit your situation. For instance, Camel provides extensions for writing routes in XML, as we'll discuss next.

2.4 Defining routes in XML

The Java DSL is certainly a more powerful option for the experienced Java developer and can lead to more-concise route definitions. But having the ability to define the same thing in XML opens a lot of possibilities. Maybe some users writing Camel routes aren't the most comfortable with Java; for example, we know many system administrators who handily write up Camel routes to solve integration problems but have never used Java in their lives. The XML configuration also makes nice graphical tooling¹ that has round-trip capabilities possible; you can edit both the XML and graphical representation of a route, and both are kept in sync. Round-trip tooling with Java is possible, but it's a seriously hard thing to do, so none is yet available.

^{1.} You can find out more about tooling options for Camel in Chapter 19.

At the time of this writing, you can write XML routes in two Inversion of Control (IoC) Java containers: Spring and OSGi Blueprint. An IoC framework allows you to “wire” beans together to form applications. This wiring is typically done through an

XML configuration file. This section gives you a quick introduction to creating applications with Spring so the IoC concept becomes clear. We'll then show you how Camel uses Spring to form a replacement or complementary solution to the Java DSL.

NOTE For a more comprehensive view of Spring, we recommend *Spring in Action* by Craig Walls (Manning, 2014). OSGi Blueprint is covered nicely in *OSGi in Action* by Richard S. Hall et al. (Manning, 2011).

The setup is certainly different between Spring and OSGi Blueprint, yet both have identical route definitions, so we cover only Spring-based examples in this chapter. Throughout the rest of the book, we refer to routes in Spring or Blueprint as just the *XML DSL*.

2.4.1 BEAN INJECTION AND SPRING

Creating an application from beans by using Spring is simple. All you need are a few Java beans (classes), a Spring XML configuration file, and `ApplicationContext`. `ApplicationContext` is similar to `camelContext`, in that it's the runtime container for Spring. Let's look at a simple example.

Consider an application that prints a greeting followed by your username. In this application, you don't want the greeting to be hardcoded, so you can use an interface to break this dependency. Consider the following interface:

```
public interface Greeter {  
    public String sayHello();  
}
```

This interface is implemented by the following classes:

```
public class EnglishGreeter implements Greeter {  
    public String sayHello() {  
        return "Hello " + System.getProperty("user.name");  
    }  
}
```

```

        }
    }
public class DanishGreeter implements Greeter {
    public String sayHello() {
        return "Davs " + System.getProperty("user.name");
    }
}

```

You can now create a greeter application as follows:

```

public class GreetMeBean {
    private Greeter greeter;

    public void setGreeter(Greeter greeter) {
        this.greeter = greeter;
    }
    public void execute() {
        System.out.println(greeter.sayHello());
    }
}

```

This application will output a different greeting depending on how you configure it. To configure this application using Spring XML, you could do something like this:

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-
beans.xsd">
    <bean id="myGreeter"
          class="camelinaction.EnglishGreeter"/>
    <bean id="greetMeBean" class="camelinaction.GreetMeBean">
        <property name="greeter" ref="myGreeter"/>
    </bean>
</beans>

```

This XML file instructs Spring to do the following:

1. Create an instance of EnglishGreeter and name the bean myGreeter
2. Create an instance of GreetMeBean and name the bean greetMeBean

- Set the reference of the greeter property of the GreetMeBean to the bean named myGreeter

This configuring of beans is called *wiring*.

To load this XML file into Spring, you can use the `ClassPathXmlApplicationContext`, which is a concrete implementation of `ApplicationContext` that's provided with the Spring framework. This class loads Spring XML files from a location specified on the classpath.

Here's the final version of `GreetMeBean`:

```
public class GreetMeBean {  
    public static void main(String[] args) {  
        ApplicationContext context =  
            new  
        ClassPathXmlApplicationContext("beans.xml");  
        GreetMeBean bean = (GreetMeBean)  
        context.getBean("greetMeBean");  
        bean.execute();  
    }  
}
```

The `ClassPathXmlApplicationContext` you instantiate here loads up the bean definitions you saw previously in the `beans.xml` file. You then call `getBean` on the context to look up the bean with the `greetMeBean` ID in the Spring registry. All beans defined in this file are accessible in this way.

To run this example, go to the `chapter2/spring` directory in the book's source code and run this Maven command:

```
mvn compile exec:java -  
Dexec.mainClass=camelinaction.GreetMeBean
```

This will output something like the following on the command line:

```
Hello janstey
```

If you had wired in `DanishGreeter` instead (that is, used the `camelinaction.DanishGreeter` class for the `myGreeter` bean), you'd have seen something like this on the console:

Davs janstey

This example may seem simple, but it should give you an understanding of what Spring and, more generally, an IoC container, really is. How does Camel fit into this? Camel can be configured as if it were another bean. Recall how you configured the JMS component to connect to an ActiveMQ broker in section 2.2.2 by using Java code:

```
ConnectionFactory connectionFactory =
    new ActiveMQConnectionFactory("vm://localhost");
CamelContext context = new DefaultCamelContext();
context.addComponent("jms",

JmsComponent.jmsComponentAutoAcknowledge(connectionFactory)
);
```

You could have done this in Spring by using the bean terminology, as follows:

```
<bean id="jms"
class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
        <bean
class="org.apache.activemq.ActiveMQConnectionFactory">
            <property name="brokerURL" value="vm://localhost"/>
        </bean>
    </property>
</bean>
```

In this case, if you send to an endpoint such as "jms:incomingOrders", Camel will look up the jms bean, and if it's of type org.apache.camel.Component, it will use that. So you don't have to manually add components to CamelContext—a task that you did manually in section 2.2.2 for the Java DSL.

But where's camelContext defined in Spring? Well, to make things easier on the eyes, Camel uses Spring extension mechanisms to provide custom XML syntax for Camel concepts within the Spring XML file. To load up camelContext in Spring, you can do the following:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
```

```

instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-
beans.xsd
        http://camel.apache.org/schema/spring
        http://camel.apache.org/schema/spring/camel-
spring.xsd">
    ...
<camelContext
xmlns="http://camel.apache.org/schema/spring"/>
</beans>
```

This automatically starts `SpringCamelContext`, which is a subclass of `DefaultCamelContext`, which you used for the Java DSL. Also notice that you have to include the `http://camel.apache.org/schema/spring/camel-spring.xsd` XML schema definition in the XML file; this is needed to import the custom XML elements.

This snippet alone isn't going to do much for you. You need to tell Camel what routes to use, as you did when using the Java DSL. The following code uses Spring XML to produce the same results as the code in [listing 2.1](#).

Listing 2.2 A Spring configuration that produces the same results as listing 2.1

```

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

    <bean id="jms"
class="org.apache.camel.component.jms.JmsComponent">
        <property name="connectionFactory">
            <bean
class="org.apache.activemq.ActiveMQConnectionFactory">
```

```

        <property name="brokerURL" value="vm://localhost"
/>
    </bean>
</property>
</bean>
<bean id="ftpToJmsRoute"
class="camelaction.FtpToJMSRoute"/>

<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <routeBuilder ref="ftpToJmsRoute"/>
</camelContext>
</beans>
```

You may have noticed that we're referring to the `camelaction.FtpToJMSRoute` class as a `RouteBuilder`. To reproduce the Java DSL example in [listing 2.1](#), you have to factor out the anonymous `RouteBuilder` into its own named class. The `FtpToJMSRoute` class looks like this:

```

public class FtpToJMSRoute extends RouteBuilder {
    public void configure() {
        from("ftp://rider.com/orders?
username=rider&password=secret")
            .to("jms:incomingOrders");
    }
}
```

Now that you know the basics of Spring and how to load Camel inside it, we can go further by looking at how to write Camel routing rules purely in XML—no Java DSL required.

2.4.2 THE XML DSL

What we've seen of Camel's integration with Spring is adequate, but it isn't taking full advantage of Spring's methodology of configuring applications using no code. To completely invert the control of creating applications using Spring XML, Camel provides custom XML extensions that we call the *XML DSL*. The XML DSL allows you to do almost everything you can do in the Java DSL.

Let's continue with the Rider Auto Parts example shown in [listing 2.2](#), but this time you'll specify the routing rules defined in

RouteBuilder purely in XML. The Spring XML in the following listing does this.

Listing 2.3 An XML DSL example that produces the same results as listing 2.1

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

    <bean id="jms"
          class="org.apache.camel.component.jms.JmsComponent">
        <property name="connectionFactory">
            <bean
                class="org.apache.activemq.ActiveMQConnectionFactory">
                <property name="brokerURL" value="vm://localhost"
            />
            </bean>
        </property>
    </bean>
    <camelContext
        xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="ftp://rider.com/orders?
username=rider&password=secret"/>
            <to uri="jms:incomingOrders"/>
        </route>
    </camelContext>
</beans>
```

In this listing, under the camelContext element, you replace routeBuilder with the route element. Within the route element, you specify the route by using elements with names similar to ones used inside the Java DSL RouteBuilder. Notice that we had to modify the FTP endpoint URI to ensure that it's valid XML. The ampersand character (&) used to define extra URI options is a reserved character in XML, so you have to escape it by using

& . With this small change, this listing is functionally equivalent to the Java DSL version in [listing 2.1](#) and the Spring plus Java DSL combo in [listing 2.2](#).

In the book's source code, we changed the `from` method to consume messages from a local file directory instead. The new route looks like this:

```
<route>
    <from uri="file:src/data?noop=true"/>
    <to uri="jms:incomingOrders"/>
</route>
```

The file endpoint will load order files from the relative `src/data` directory. The `noop` property configures the endpoint to leave the file as is after processing; this option is useful for testing. In chapter 6, you'll see how Camel allows you to delete or move the files after processing.

This route won't display anything interesting yet. You need to add a processing step for testing.

ADDING A PROCESSOR

Adding processing steps is simple, as in the Java DSL. Here you'll add a custom processor as you did in section 2.3.2.

Because you can't refer to an anonymous class in Spring XML, you need to factor out the anonymous processor into the following class:

```
public class DownloadLogger implements Processor {
    public void process(Exchange exchange) throws Exception
    {
        System.out.println("We just downloaded: "
                           +
                           exchange.getIn().getHeader("CamelFileName"));
    }
}
```

You can now use the processor in your XML DSL route as follows:

```
<bean id="downloadLogger"
  class="camelinaction.DownloadLogger"/>

<camelContext
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file:src/data?noop=true"/>
    <process ref="downloadLogger"/>
    <to uri="jms:incomingOrders"/>
  </route>
</camelContext>
```

Now you're ready to run the example. Go to the chapter2/spring directory in the book's source code and run this Maven command:

```
mvn clean compile camel:run
```

Because there's only one message file named message1.xml in the src/data directory, this outputs something like the following on the command line:

```
We just downloaded: message1.xml
```

What if you wanted to print this message after consuming it from the incomingOrders queue? To do this, you need to create another route.

USING MULTIPLE ROUTES

You may recall that in the Java DSL each Java statement starting with a `from` creates a new route. You can also create multiple routes with the XML DSL. To do this, add a `route` element within the `camelContext` element.

For example, move the `DownloadLogger` processor into a second route, after the order gets sent to the `incomingOrders` queue:

```
<camelContext
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file:src/data?noop=true"/>
    <to uri="jms:incomingOrders"/>
  </route>
```

```
<route>
    <from uri="jms:incomingOrders"/>
    <process ref="downloadLogger"/>
</route>
</camelContext>
```

Now you're consuming the message from the `incomingOrders` queue in the second route, so the downloaded message will be printed after the order is sent via the queue.

CHOOSING WHICH DSL TO USE

Which DSL is best to use in a particular scenario is a common question for Camel users, but it mostly comes down to personal preference. If you like working with Spring or like defining things in XML, you may prefer a pure XML approach. If you want to be hands-on with Java, maybe a pure Java DSL approach is better for you.

In either case, you'll be able to access nearly all of Camel's functionality. The Java DSL is a slightly richer language to work with because you have the full power of the Java language at your fingertips. Also, some Java DSL features, such as value builders (for building expressions and predicates), aren't available in the XML DSL. On the other hand, using Spring XML gives you access to the wonderful object construction capabilities as well as commonly used Spring abstractions for things like database connections and JMS integration. The XML DSL also makes nice graphical tooling possible that has round-trip capabilities: you can edit both the XML and graphical representation of a route, and both are kept in sync.² A common compromise is to use both Spring XML and the Java DSL, which is one of the topics we'll cover next.

^{2.} See Bilgin Ibryam's blog post on which Camel DSL to use:
<http://www.ofbizian.com/2017/12/which-camel-dsl-to-choose-and-why.html>.

2.4.3 USING CAMEL AND SPRING

Whether you write your routes in the Java or XML DSL, running

Camel in a Spring container gives you many other benefits. For one, if you're using the XML DSL, you don't have to recompile any code when you want to change your routing rules. Also, you gain access to Spring's portfolio of database connectors, transaction support, and more.

Let's take a closer look at what other Spring integrations Camel provides.

FINDING ROUTE BUILDERS

Using the Spring `camelContext` as a runtime and the Java DSL for route development is a great way to use Camel. You saw before in [listing 2.2](#) that you can explicitly tell the Spring `camelContext` what route builders to load. You can do this by using the `routeBuilder` element:

```
<camelContext
  xmlns="http://camel.apache.org/schema/spring">
  <routeBuilder ref="ftpToJmsRoute"/>
</camelContext>
```

Being this explicit results in a clean and concise definition of what is being loaded into Camel.

Sometimes, though, you may need to be a bit more dynamic. This is where the `packageScan` and `contextScan` elements come in:

```
<camelContext
  xmlns="http://camel.apache.org/schema/spring">
  <packageScan>
    <package>camelinaction.routes</package>
  </packageScan>
</camelContext>
```

This `packageScan` element will load all `RouteBuilder` classes found in the `camelinaction.routes` package, including all subpackages.

You can even be a bit pickier about what route builders are included:

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

```
<packageScan>
    <package>camelaction.routes</package>
    <excludes>**.*Test*</excludes>
    <includes>**.*</includes>
</packageScan>
</camelContext>
```

In this case, you're loading all route builders in the `camelaction.routes` package, except for ones with `Test` in the class name. The matching syntax is similar to what's used in Apache Ant's file pattern matchers.

The `contextScan` element takes advantage of Spring's component-scan feature to load any Camel route builders that are marked with the `org.springframework.stereotype.Component` annotation. Let's modify the `FtpToJMSRoute` class to use this annotation:

```
@Component
public class FtpToJMSRoute extends RouteBuilder {
    public void configure() {
        from("ftp://rider.com" +
            "/orders?username=rider&password=secret")
            .to("jms:incomingOrders");
    }
}
```

You can now enable the component scanning by using the following configuration in your Spring XML file:

```
<context:component-scan base-
package="camelaction.routes"/>
<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <contextScan/>
</camelContext>
```

This will load up any Camel route builders within the `camelaction.routes` package that have the `@Component` annotation.

Under the hood, some of Camel's components, such as the JMS component, are built on top of abstraction libraries from Spring. This often explains why configuring those components is

easy in Spring.

CONFIGURING COMPONENTS AND ENDPOINTS

You saw in section 2.4.1 that components could be defined in Spring XML and would be picked up automatically by Camel. For instance, look at the JMS component again:

```
<bean id="jms"
  class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean
      class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://localhost"/>
    </bean>
  </property>
</bean>
```

The bean `id` defines what this component will be called. This gives you the flexibility to give the component a more meaningful name based on the use case. Your application may require the integration of two JMS brokers, for instance. One could be for Apache ActiveMQ and another could be for WebSphere MQ:

```
<bean id="activemq"
  class="org.apache.camel.component.jms.JmsComponent">
  ...
</bean>
<bean id="wmq"
  class="org.apache.camel.component.jms.JmsComponent">
  ...
</bean>
```

You could then use URIs such as `activemq:myActiveMQQueue` or `wmq:myWebSphereQueue`. Endpoints can also be defined by using Camel's Spring XML extensions. For example, you can break out the FTP endpoint for connecting to the Rider Auto Parts legacy order server into an `<endpoint>` element that's highlighted in bold here:

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

```
<endpoint id="ridersFtp"
    uri="ftp://rider.com/orders?
username=rider&password=secret"/>
<route>
    <from ref="ridersFtp"/>
    <to uri="jms:incomingOrders"/>
</route>
</camelContext>
```

NOTE You may notice that credentials have been added directly into the endpoint URI, which isn't always the best solution. A better way is to refer to credentials that are defined and sufficiently protected elsewhere. In section 14.1 of chapter 14, you can see how the Camel Properties component or Spring property placeholders are used to do this.

For longer endpoint URIs, it's often easier to read them if you break them up over several lines. See the previous route with the endpoint URI options broken into separate lines:

```
<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <endpoint id="ridersFtp"
uri="ftp://rider.com/orders?

username=rider&

password=secret"/>
    <route>
        <from ref="ridersFtp"/>
        <to uri="jms:incomingOrders"/>
    </route>
</camelContext>
```

IMPORTING CONFIGURATION AND ROUTES

A common practice in Spring development is to separate an application's wiring into several XML files. This is mainly done to make the XML more readable; you probably wouldn't want to wade through thousands of lines of XML in a single file without some separation.

Another reason to separate an application into several XML files is the potential for reuse. For instance, another application may require a similar JMS setup, so you can define a second Spring XML file called `jms-setup.xml` with these contents:

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

    <bean id="jms"
          class="org.apache.camel.component.jms.JmsComponent">
        <property name="connectionFactory">
            <bean
                class="org.apache.activemq.ActiveMQConnectionFactory">
                <property name="brokerURL" value="vm://localhost"
            />
            </bean>
        </property>
    </bean>
</beans>
```

This file could then be imported into the XML file containing `CamelContext` by using the following line:

```
<import resource="jms-setup.xml"/>
```

Now `CamelContext` can use the JMS component configuration even though it's defined in a separate file.

Other useful things to define in separate files are the XML DSL routes themselves. Because route elements need to be defined within a `camelContext` element, an additional concept is introduced to define routes. You can define routes within a `routeContext` element, as shown here:

```
<routeContext id="ftpToJms"
              xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="ftp://rider.com/orders?
username=rider&password=secret"/>
        <to uri="jms:incomingOrders"/>
```

```
</route>  
</routeContext>
```

This `routeContext` element could be in another file or in the same file. You can then import the routes defined in this `routeContext` with the `routeContextRef` element. You use the `routeContextRef` element inside `camelContext` as follows:

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
    <routeContextRef ref="ftpToJms"/>  
</camelContext>
```

If you import `routeContext` into multiple `CamelContexts`, a new instance of the route is created in each. In the preceding case, two identical routes, with the same endpoint URIs, will lead to them competing for the same resource. In this case, only one route at a time will receive a particular file from FTP. In general, you should take care when reusing routes in multiple `CamelContexts`.

SETTING ADVANCED SPRING CONFIGURATION OPTIONS

Many other configuration options are available when using the Spring `CamelContext`:

- Pluggable bean registries are discussed in chapter 4.
- The configuration of interceptors is covered in chapter 9.
- Stream caching, fault handling and startup are mentioned in chapter 15.
- The Tracer mechanism is covered in chapter 16.

2.5 *Endpoints revisited*

In section 2.2, we covered the basics of endpoints in Camel. Now that you've seen both Java and XML routes in action, it's time to introduce more-advanced endpoint configurations.

2.5.1 SENDING TO DYNAMIC ENDPOINTS

Endpoint URIs like the JMS one shown in section 2.2.2 are evaluated just once when Camel starts up, so they're static entities in Camel. This is fine for most scenarios, when you know ahead of time what the destination names will be called. But what if you need to determine these names at runtime? Static endpoint URIs provided to the `to` method will be of no use in this case, because they're evaluated only once at startup. Camel provides an additional DSL method for this: `toD`.

For example, say you want to make the endpoint URI point to a destination name stored as a message header. The following does that:

```
.toD("jms:queue:${header.myDest}");
```

And in the XML DSL:

```
<toD uri="jms:queue:${header.myDest}"/>
```

This endpoint URI uses a Simple expression within the `${ }` placeholders to return the value of the `myDest` header in the incoming message. If `myDest` is `incomingOrders`, the resulting endpoint URI will be `jms:queue:incomingOrders`, as we had before in the static case. If you were wondering about the Simple language, it's a lightweight expression language built into Camel's core. We go over Simple in more detail in section 2.6.1 of this chapter, and appendix A provides a complete reference.

To run this example yourself, go to the `chapter2/ftp-jms` directory in the book's source code and run this Maven command:

```
mvn test -Dtest=FtpToJMSWithDynamicToTest
```

2.5.2 USING PROPERTY PLACEHOLDERS IN ENDPOINT URIS

Rather than having hardcoded endpoint URIs, Camel allows you to use property placeholders in the URIs to replace the dynamic

parts. By *dynamic*, we mean that values will be replaced when Camel starts up, not on every new message, as in the case of `toD` described in the previous section.

One common usage of property placeholders is in testing. A Camel route is often tested in different environments—you may want to test it locally on your laptop, and then later on a dedicated test platform, and so forth. But you don’t want to rewrite tests every time you move to a new environment. That’s why you externalize dynamic parts rather than hardcoding them.

USING THE PROPERTIES COMPONENT

Camel has a Properties component to support externalizing properties defined in the routes (and elsewhere). The Properties component works in much the same way as Spring property placeholders, but it has a few noteworthy improvements:

- It’s built in the camel-core JAR, which means it can be used without the need for Spring or any third-party framework.
- It can be used in all the DSLs, such as the Java DSL, and isn’t limited to Spring XML files.
- It supports masking sensitive information by plugging in third-party encryption libraries.

For more details on the Properties component, see the Camel documentation: <http://camel.apache.org/properties.html>.

TIP You can use the Jasypt component to encrypt sensitive information in the properties file. For example, you may not want to have passwords in clear text in the properties file. You can read more about the Jasypt component in chapter 14.

To ensure that the property placeholder is loaded and in use as early as possible, you have to configure `PropertiesComponent` when `CamelContext` is created:

```
CamelContext context = new DefaultCamelContext();
PropertiesComponent prop = camelContext.getComponent(
    "properties", PropertiesComponent.class);
prop.setLocation("classpath:rider-test.properties");
```

In the rider-test.properties file, you define the externalized properties as key-value pairs:

```
myDest=incomingOrders
```

RouteBuilders can then take advantage of the externalized properties directly in the endpoint URI, as shown in bold in this route:

```
return new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("file:src/data?noop=true")
            .to("jms:{myDest}");
        from("jms:incomingOrders")
            .to("mock:incomingOrders");
    }
};
```

You should notice that the Camel syntax for property placeholders is a bit different than for Spring property placeholders. The Camel Properties component uses the {{key}} syntax, whereas Spring uses \${key}.

You can try this example by using the following Maven goal from the chapter2/ftp-jms directory:

```
mvn test -Dtest=FtpToJMSWithPropertyPlaceholderTest
```

Setting this up in XML is a bit different, as you'll see in the next section.

USING PROPERTY PLACEHOLDERS IN THE XML DSL

To use the Camel Properties component in Spring XML, you have to declare it as a Spring bean with the ID properties, as shown here:

```
<bean id="properties"
```

```
class="org.apache.camel.component.properties.PropertiesComponent">
    <property name="location" value="classpath:rider-test.properties"/>
</bean>
```

In the rider-test.properties file, you define the externalized properties as key-value pairs:

```
myDest=incomingOrders
```

The `camelContext` element can then take advantage of the externalized properties directly in the endpoint URI, as shown in bold in this route:

```
<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file:src/data?noop=true"/>
        <to uri="jms:{myDest}" />
    </route>
    <route>
        <from uri="jms:incomingOrders"/>
        <to uri="mock:incomingOrders"/>
    </route>
</camelContext>
```

Instead of using a Spring bean to define the Camel Properties component, you can also use a specialized `<propertyPlaceholder>` within `camelContext`, as follows:

```
<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <propertyPlaceholder id="properties"
        location="classpath:rider-test.properties"/>
    <route>
        <from uri="file:src/data?noop=true"/>
        <to uri="jms:{myDest}" />
    </route>
    <route>
        <from uri="jms:incomingOrders"/>
        <to uri="mock:incomingOrders"/>
    </route>
</camelContext>
```

This example is included in the book's source code in the chapter2/spring directory. You can try it by using the following Maven goal:

```
mvn test -Dtest=SpringFtpToJMSWithPropertyPlaceholderTest
```

We'll now cover the same example, but using Spring property placeholders instead of the Camel Properties component.

USING SPRING PROPERTY PLACEHOLDERS

The Spring Framework supports externalizing properties defined in the Spring XML files by using a feature known as Spring *property placeholders*. We'll review the example from the previous section, using Spring property placeholders instead of the Camel Properties component.

The first thing you need to do is set up the route having the endpoint URIs externalized. This could be done as follows. Notice that Spring uses the \${key} syntax:

```
<context:property-placeholder properties-
ref="properties"/>
<util:properties id="properties"
                 location="classpath:rider-
test.properties"/>

<camelContext
xmlns="http://camel.apache.org/schema/spring">

    <endpoint id="myDest" uri="jms:${myDest}" />

    <route>
        <from uri="file:src/data?noop=true"/>
        <to uri="jms:${myDest}" />
    </route>

    <route>
        <from uri="jms:incomingOrders"/>
        <to uri="mock:incomingOrders" />
    </route>
</camelContext>
```

Unfortunately, the Spring Framework doesn't support using

placeholders directly in endpoint URIs in the route, so you must define endpoints that include those placeholders by using the `<endpoint>` tag. The following code snippet shows how this is done:

```
<context:property-placeholder properties-  
ref="properties"/>  
  <util:properties id="properties" ❶  
location="classpath:rider-test.properties"/>
```

❶

Loads properties from external file

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
  
  <endpoint id="myDest" uri="jms:${myDest}"> ❷
```

❷

Defines endpoint using Spring property placeholders

```
<route>  
  <from uri="file:src/data?noop=true"/>  
  <to ref="myDest"> ❸
```

❸

Refers to endpoint in route

```
</route>  
  
<route>  
  <from uri="jms:incomingOrders"/>  
  <to uri="mock:incomingOrders"/>  
</route>  
</camelContext>
```

To use Spring property placeholders, you must declare the `<context:property-placeholder>` tag where you refer to a properties bean ❶ that will load the properties file from the

classpath.

In the `camelContext` element, you define an endpoint ❷ that uses a placeholder for a dynamic JMS destination name. The `${myDest}` is a Spring property placeholder that refers to a property with the key `myDest`.

In the route, you must refer to the endpoint ❸ instead of using the regular URI notations. Notice the use of the `ref` attribute in the `<to>` tag.

The `rider-test.properties` properties file contains the following line:

```
myDest=incomingOrders
```

This example is included in the book's source code in the `chapter2/spring` directory. You can try it by using the following Maven goal:

```
mvn test -Dtest=SpringFtpToJMSwithSpringPropertyPlaceholderTest
```

The Camel Properties component vs. Spring property placeholders

The Camel Properties component is more powerful than the Spring property placeholder mechanism. The latter works only when defining routes using Spring XML, and you have to declare the endpoints in dedicated `<endpoint>` tags for the property placeholders to work.

The Camel Properties component is provided out of the box, which means you can use it without using Spring at all. And it supports the various DSL languages you can use to define routes, such as Java, Spring XML, and Blueprint OSGi XML. On top of that, you can declare the placeholders anywhere in the route definitions.

2.5.3 USING RAW VALUES IN ENDPOINT URIS

Sometimes values you want to use in a URI will make the URI itself invalid. Take, for example, an FTP password of `++%%w?rd`. Adding this as is would break any URI because it uses reserved characters. You could encode these reserved characters, but that would make things less readable (not that you'd want your password more readable—it's just an example!). Camel's solution is to allow “raw” values in endpoint URIs that don't count toward URI validation. For example, let's use this raw password to connect to the `rider.com` FTP server:

```
from("ftp://rider.com/orders?  
username=rider&password=RAW(++%%w?rd)")
```

As you can see, the password is surrounded by `RAW()`, which will make Camel treat this value as a raw value.

2.5.4 REFERENCING REGISTRY BEANS IN ENDPOINT URIS

You've heard the Camel registry mentioned a few times now but haven't seen it in use. We don't dive into detail about the registry here (that's covered in chapter 4), but we'll show a common syntax in endpoint URIs related to the registry. Anytime a Camel endpoint requires an object instance as an option value, you can refer to one in the registry by using the `#` syntax. For example, say you want to fetch only CSV order files from the FTP site. You could define a filter like so:

```
public class OrderFileFilter<T> implements  
GenericFileFilter<T> {  
    public boolean accept(GenericFile<T> file) {  
        return file.getFileName().endsWith("csv");  
    }  
}
```

Add it to the registry:

```
registry.bind("myFilter", new OrderFileFilter<Object>());
```

Then you use the `#` syntax to refer to the named instance in the

registry:

```
from("ftp://rider.com/orders?  
username=rider&password=secret&filter=#myFilter")
```

With these endpoint configuration techniques behind you, you're ready to tackle more-advanced routing topics by using Camel's implementation of the EIPs.

2.6 Routing and EIPs

So far, we haven't touched much on the EIPs that Camel was built to implement. That's intentional. We want to make sure you have a good understanding of what Camel is doing in the simplest cases before moving on to more-complex examples.

As far as EIPs go, we'll be looking at the Content-Based Router, Message Filter, Multicast, Recipient List, and Wire Tap right away. Other patterns are introduced throughout the book, and chapter 5 covers the most complex EIPs. The complete list of EIPs supported by Camel is available from the Camel website (<http://camel.apache.org/eip.html>).

For now, let's start by looking at the most well-known EIP: the Content-Based Router.

2.6.1 USING A CONTENT-BASED ROUTER

As the name indicates, a *content-based router* (CBR) is a message router that routes a message to a destination based on its content. The content could be a message header, the payload data type, or part of the payload itself—pretty much anything in the message exchange.

To demonstrate, let's go back to Rider Auto Parts. Some customers have started uploading orders to the FTP server in the newer XML format rather than CSV. You have two types of messages coming in to the `incomingOrders` queue. We didn't touch on this before, but you need to convert the incoming orders into an internal POJO format. You need to do different

conversions for the different types of incoming orders.

As a possible solution, you could use the filename extension to determine whether a particular order message should be sent to a queue for CSV orders or a queue for XML orders. This is depicted in [figure 2.10](#).

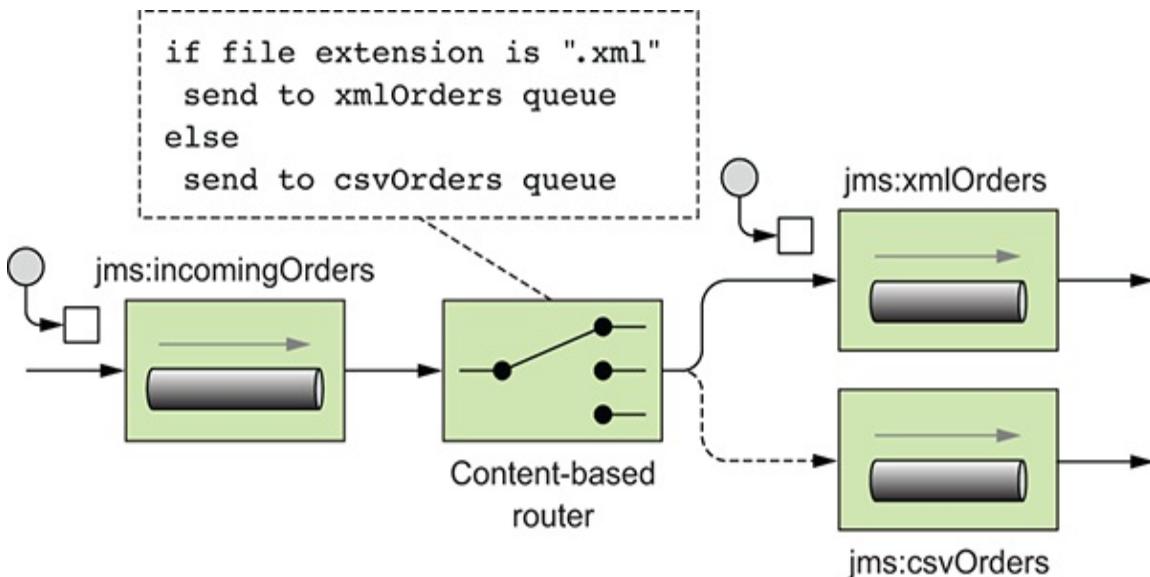


Figure 2.10 The CBR routes messages based on their content. In this case, the filename extension (as a message header) is used to determine which queue to route to.

As you saw earlier, you can use the `CamelFileName` header set by the FTP consumer to get the filename.

To do the conditional routing required by the CBR, Camel introduces a few keywords in the DSL. The `choice` method creates a CBR processor, and conditions are added by following `choice` with a combination of a `when` method and a predicate.

Camel's creators could have chosen `contentBasedRouter` for the method name, to match the EIP, but they stuck with `choice` because it reads more naturally. It looks like this:

```
from("jms:incomingOrders")
    .choice()
        .when(predicate)
            .to("jms:xmlOrders")
        .when(predicate)
            .to("jms:csvOrders");
```

You may have noticed that we didn't fill in the predicates required for each `when` method. A *predicate* in Camel is a simple interface that has only a `matches` method:

```
public interface Predicate {  
    boolean matches(Exchange exchange);  
}
```

For example, you can think of a predicate as a `boolean` condition in a Java `if` statement.

You probably don't want to look inside the exchange yourself and do a comparison. Fortunately, predicates are often built up from expressions, and expressions are used to extract a result from an exchange based on the expression content. You can choose from many expression languages in Camel, some of which include Simple, SpEL, JXPath, MVEL, OGNL, JavaScript, Groovy, XPath, and XQuery. As you'll see in chapter 4, you can even use a method call to a bean as an expression in Camel. In this case, you'll be using the expression builder methods that are part of the Java DSL.

Within `RouteBuilder`, you can start by using the `header` method, which returns an expression that will evaluate to the header value. For example, `header("CamelFileName")` creates an expression that will resolve to the value of the `CamelFileName` header on the incoming exchange. On this expression, you can invoke methods to create a predicate. To check whether the filename extension is equal to `.xml`, you can use the following predicate:

```
header("CamelFileName").endsWith(".xml")
```

The completed CBR is shown in the following listing.

Listing 2.4 A complete content-based router using the Java DSL

```
return new RouteBuilder() {  
    @Override  
    public void configure() throws Exception {
```

```
// load file orders from src/data into the JMS
queue
    from("file:src/data?
noop=true").to("jms:incomingOrders");

from("jms:incomingOrders") 1
```

1

Content-based router

```
.choice() 1
.when(header("CamelFileName").endsWith(".xml")) 1
    .to("jms:xmlOrders") 1

.when(header("CamelFileName").endsWith(".csv")) 1
    .to("jms:csvOrders"); 1

from("jms:xmlOrders") 2
```

2

Test routes that print message content

```
.log("Received XML order:
${header.CamelFileName}") 2
    .to("mock:xml") 2

from("jms:csvOrders") 2
    .log("Received CSV order:
${header.CamelFileName}") 2
        .to("mock:csv"); 2
    }
};
```

To run this example, go to the chapter2/cbr directory in the book's source code and run this Maven command:

```
mvn test -Dtest=OrderRouterTest
```

This consumes two order files in the chapter2/cbr/src/data directory and outputs the following:

```
Received CSV order: message2.csv
Received XML order: message1.xml
```

The output comes from the two routes at the end of the configure method ②. These routes consume messages from the `xmlOrders` and `csvOrders` queues and then print messages by using the `log` DSL method.

The Simple language

You may have noticed that the string passed into the `log` method has something that looks like properties defined for expansion. This `${header.CamelFileName}` term is from the Simple language, which is Camel's own expression language included with the camel-core module. The Simple language contains many useful variables, functions, and operators that operate on the incoming exchange. A dynamic expression in the Simple language is enclosed with the `${}` placeholders, as you saw in [listing 2.4](#). Let's consider this example:

```
 ${header.CamelFileName}
```

Here, `header` maps to the headers of the `in` message of the exchange. After the dot, you can append any header name that you want to access. At runtime, Camel will return the value of the `CamelFileName` header in the `in` message. You could also replace your content-based router conditions in [listing 2.4](#) with simple expressions:

```
from("jms:incomingOrders")
    .choice()
        .when(simple("${header.CamelFileName} ends
with 'xml'"))
            .to("jms:xmlOrders")
        .when(simple("${header.CamelFileName} ends
with 'csv'"))
            .to("jms:csvOrders");
```

Here you use the `ends with` operator to check the end of the string returned from the `${header.CamelFileName}` dynamic simple expression. The Simple language is so

useful for Camel applications that we devote appendix A to cover it fully.

You use these routes to test that the router ❶ is working as expected. Route-testing techniques, like use of the Mock component, are discussed in chapter 9.

You can also form an equivalent CBR by using the XML DSL, as shown in [listing 2.5](#). Other than being in XML rather than Java, the main difference is that you use a Simple expression instead of the Java-based predicate ❶. The Simple expression language is a great option for replacing predicates from the Java DSL.

Listing 2.5 A complete content-based router using the XML DSL

```
<route>
  <from uri="file:src/data?noop=true"/>
  <to uri="jms:incomingOrders"/>
</route>

<route>
  <from uri="jms:incomingOrders"/>
  <choice>
    <when>
      <simple>${header.CamelFileName} ends with
'xml'</simple> ❶
    <to uri="jms:xmlOrders"/>
  </when>
  <when>
    <simple>${header.CamelFileName} ends with
'csv'</simple> ❶
    <to uri="jms:csvOrders"/>
  </when>
  </choice>
</route>
```

❶

Simple expression used instead of Java-based predicate

```
    <to uri="jms:xmlOrders"/>
  </when>
  <when>
    <simple>${header.CamelFileName} ends with
'csv'</simple> ❶
    <to uri="jms:csvOrders"/>
  </when>
  </choice>
</route>
```

```
<route> ②
```

Test routes that print message content

```
<from uri="jms:xmlOrders"/> ②
<log message="Received XML order:
${header.CamelFileName}"/> ②
<to uri="mock:xml"/> ②
</route> ②
```

```
<route> ②
<from uri="jms:csvOrders"/> ②
<log message="Received CSV order:
${header.CamelFileName}"/> ②
<to uri="mock:csv"/> ②
</route> ②
```

To run this example, go to the chapter2/cbr directory in the book's source code and run this Maven command:

```
mvn test -Dtest=SpringOrderRouterTest
```

You'll see output similar to that of the Java DSL example.

USING THE OTHERWISE CLAUSE

A Rider Auto Parts customer sends CSV orders with the .csl extension. Your current route handles only .csv and .xml files and will drop all orders with other extensions. This isn't a good solution, so you need to improve things a bit.

One way to handle the extra extension is to use a regular expression as a predicate instead of the `endsWith` call. The following route can handle the extra file extension:

```
from("jms:incomingOrders")
.choice()
    .when(header("CamelFileName").endsWith(".xml"))
        .to("jms:xmlOrders")
    .when(header("CamelFileName").regex("^.*"
(csv|csl)$"))
        .to("jms:csvOrders");
```

This solution still suffers from the same problem, though. Any orders not conforming to the file extension scheme will be dropped. You should be handling bad orders that come in so someone can fix the problem. For this, you can use the `otherwise` clause:

```
from("jms:incomingOrders")
    .choice()
        .when(header("CamelFileName").endsWith(".xml"))
            .to("jms:xmlOrders")
        .when(header("CamelFileName").regex("^.*"
(csv|csl)$"))
            .to("jms:csvOrders")
        .otherwise()
            .to("jms:badOrders");
```

Now, all orders not having an extension of .csv, .csl, or .xml are sent to the `badOrders` queue for handling.

The equivalent route in XML DSL is as follows:

```
<route>
    <from uri="jms:incomingOrders"/>
    <choice>
        <when>
            <simple>${header.CamelFileName} ends with
'.xml'</simple>
            <to uri="jms:xmlOrders"/>
        </when>
        <when>
            <simple>${header.CamelFileName} regex '^.*'
(csv|csl)$'</simple>
            <to uri="jms:csvOrders"/>
        </when>
        <otherwise>
            <to uri="jms:badOrders"/>
        </otherwise>
    </choice>
</route>
```

To run this example, go to the `chapter2/cbr` directory in the book's source and run one or both of these Maven commands:

```
mvn test -Dtest=OrderRouterOtherwiseTest
mvn test -Dtest=SpringOrderRouterOtherwiseTest
```

This consumes four order files in the chapter2/cbr/src/data_full directory and outputs the following:

```
Received CSV order: message2.csv
Received XML order: message1.xml
Received bad order: message4.bad
Received CSV order: message3.csv
```

You can now see that a bad order has been received.

ROUTING AFTER A CONTENT-BASED ROUTER

The CBR may seem like it's the end of the route; messages are routed to one of several destinations, and that's it. Continuing the flow means you need another route, right?

Well, there are several ways you can continue routing after a CBR. One is by using another route, as you did in [listing 2.4](#) for printing a test message to the console. Another way of continuing the flow is by closing the choice block and adding another processor to the pipeline after that.

You can close the choice block by using the end method:

```
from("jms:incomingOrders")
    .choice()
        .when(header("CamelFileName").endsWith(".xml"))
            .to("jms:xmlOrders")
        .when(header("CamelFileName").regex("^.*"
(csv|csl)$"))
            .to("jms:csvOrders")
        .otherwise()
            .to("jms:badOrders")
    .end()
.to("jms:continuedProcessing");
```

Here, the choice has been closed and another to has been added to the route. After each destination with the choice, the message will be routed to the continuedProcessing queue as well. This is illustrated in [figure 2.11](#).

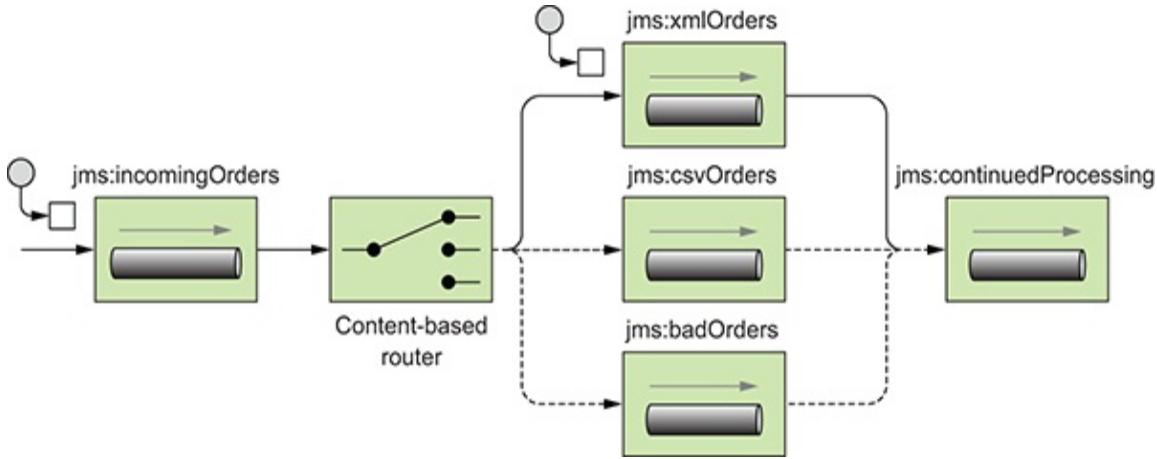


Figure 2.11 By using the `end` method, you can route messages to a destination after the CBR.

You can also control what destinations are final in the choice block. For instance, you may not want bad orders continuing through the rest of the route. You'd like them to be routed to the `badOrders` queue and stop there. In that case, you can use the `stop` method in the DSL:

```

from("jms:incomingOrders")
    .choice()
        .when(header("CamelFileName").endsWith(".xml"))
            .to("jms:xmlOrders")
        .when(header("CamelFileName").regex("^.*"(csv|csl)$"))
            .to("jms:csvOrders")
        .otherwise()
            .to("jms:badOrders").stop()
    .end()
    .to("jms:continuedProcessing");
  
```

Now, any orders entering into the `otherwise` block will be sent only to the `badOrders` queue—not to the `continuedProcessing` queue.

Using the XML DSL, this route looks a bit different:

```

<route>
    <from uri="jms:incomingOrders"/>
    <choice>
        <when>
            <simple>${header.CamelFileName} ends with
'.xml'</simple>
        
```

```

        <to uri="jms:xmlOrders"/>
    </when>
    <when>
        <simple>${header.CamelFileName} regex '^.*'
(csv|csl)$'</simple>
        <to uri="jms:csvOrders"/>
    </when>
    <otherwise>
        <to uri="jms:badOrders"/>
        <stop/>
    </otherwise>
</choice>
<to uri="jms:continuedProcessing"/>
</route>

```

Note that you don't have to use an `end()` call to end the choice block because XML requires an explicit *end block* in the form of the closing element `</choice>`.

2.6.2 USING MESSAGE FILTERS

Rider Auto Parts now has a new issue: its QA department has expressed the need to be able to send test orders into the live web front end of the order system. Your current solution would accept these orders as real and send them to the internal systems for processing. You've suggested that QA should be testing on a development clone of the real system, but management has shot down this idea, citing a limited budget. What you need is a solution that will discard these test messages while still operating on the real orders.

The Message Filter EIP, shown in figure 2.12, provides a nice way of dealing with this kind of problem. Incoming messages pass through the filter only if a certain condition is met. Messages failing the condition are dropped.

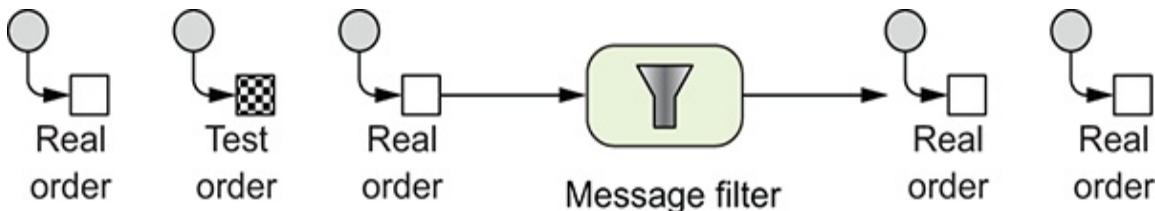


Figure 2.12 A message filter allows you to filter out uninteresting messages based on a certain condition. In this case, test messages are filtered out.

Let's see how to implement this using Camel. Recall that the web front end that Rider Auto Parts uses sends orders only in the XML format, so you can place this filter after the `xmlOrders` queue, where all orders are XML. Test messages have an extra `test` attribute set, so you can use this to do the filtering. A test message looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<order name="motor" amount="1" customer="foo" test="true"/>
```

The entire solution is implemented in `OrderRouterWithFilterTest.java`, which is included with the `chapter2/filter` project in the book's source distribution. The filter looks like this:

```
from("jms:xmlOrders")
    .filter(xpath("/order[not(@test)]"))
    .log("Received XML order: ${header.CamelFileName}")
    .to("mock:xml");
```

To run this example, execute the following Maven command on the command line:

```
mvn test -Dtest=OrderRouterWithFilterTest
```

This outputs the following on the command line:

```
Received XML order: message1.xml
```

You'll receive only one message after the filter because the test message was filtered out.

You may have noticed that this example filters out the test message with an XPath expression. XPath expressions are useful for creating conditions based on XML payloads. In this case, the expression will evaluate to `true` for orders that don't have the `test` attribute.

A message filter route in the XML DSL looks like this:

```
<route>
    <from uri="jms:xmlOrders"/>
    <filter>
        <xpath>/order[not(@test)]</xpath>
```

```
<log message="Received XML order:  
${header.CamelFileName}"/>  
    <to uri="mock:xml"/>  
    </filter>  
</route>
```

To run the XML version of the example, execute the following Maven command on the command line:

```
mvn test -Dtest=SpringOrderRouterWithFilterTest
```

So far, the EIPs you've looked at sent messages to only a single destination. Next you'll look at how to send to multiple destinations.

2.6.3 USING MULTICASTING

Often in enterprise applications you'll need to send a copy of a message to several destinations for processing. When the list of destinations is known ahead of time and is static, you can add an element to the route that will consume messages from a source endpoint and then send the message out to a list of destinations. Borrowing terminology from computer networking, we call this the Multicast EIP.

Currently at Rider Auto Parts, orders are processed in a step-by-step manner. They're first sent to accounting for validation of customer standing and then to production for manufacture. A bright new manager has suggested improving the speed of operations by sending orders to accounting and production at the same time. This would cut out the delay involved when production waits for the okay from accounting. You've been asked to implement this change to the system.

Using a multicast, you could envision the solution shown in figure 2.13.

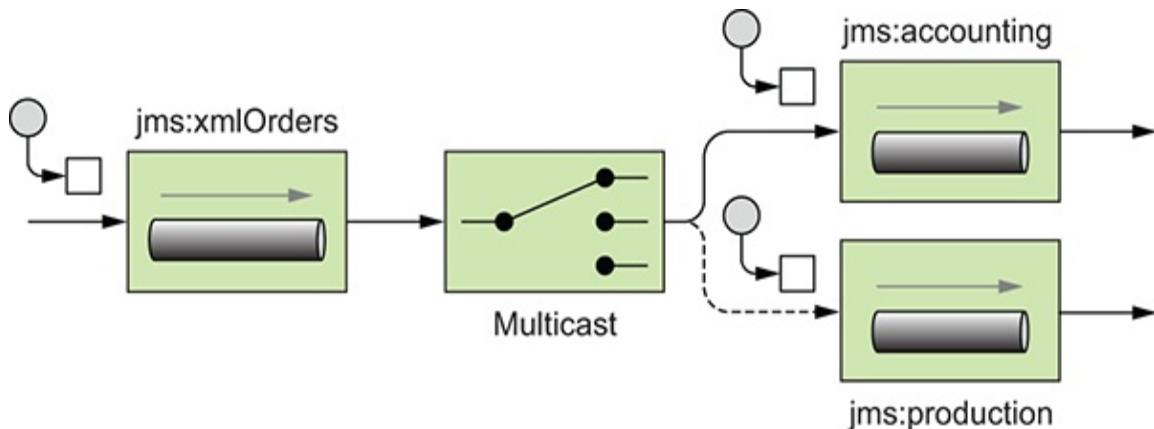


Figure 2.13 A multicast sends a message to numerous specified recipients.

With Camel, you can use the `multicast` method in the Java DSL to implement this solution:

```
from("jms:xmlOrders").multicast().to("jms:accounting",
"jms:production");
```

The equivalent route in XML DSL is as follows:

```
<route>
    <from uri="jms:xmlOrders"/>
    <multicast>
        <to uri="jms:accounting"/>
        <to uri="jms:production"/>
    </multicast>
</route>
```

To run this example, go to the `chapter2/multicast` directory in the book's source code and run these commands:

```
mvn clean test -Dtest=OrderRouterWithMulticastTest
mvn clean test -Dtest=SpringOrderRouterWithMulticastTest
```

You should see the following output on the command line:

```
Accounting received order: message1.xml
Production received order: message1.xml
```

These two lines of output are coming from two test routes that consume from the accounting and production queues and then output text to the console that qualifies the message.

TIP For dealing with responses from services invoked in a multicast, an aggregator is used. See more about aggregation in chapter 5.

By default, the multicast sends message copies sequentially. In the preceding example, a message is sent to the accounting queue and then to the production queue. But what if you want to send them in parallel?

USING PARALLEL MULTICASTING

Sending messages in parallel by using the multicast involves only one extra DSL method: `parallelProcessing`. Extending the previous multicast example, you can add the `parallelProcessing` method as follows:

```
from("jms:xmlOrders")
    .multicast().parallelProcessing()
    .to("jms:accounting", "jms:production");
```

This sets up the multicast to distribute messages to the destinations in parallel. Under the hood, a thread pool is used to manage threads. This can be replaced or configured as you see fit. For more information on the Camel threading model and thread pools, see chapter 13.

The equivalent route in XML DSL is as follows:

```
<route>
    <from uri="jms:xmlOrders"/>
    <multicast parallelProcessing="true">
        <to uri="jms:accounting"/>
        <to uri="jms:production"/>
    </multicast>
</route>
```

The main difference from the Java DSL is that the methods used to set flags such as `parallelProcessing` in the Java DSL are now attributes on the `multicast` element. To run this example, go to the `chapter2/multicast` directory in the book's source code and

run these commands:

```
mvn clean test -Dtest=OrderRouterWithParallelMulticastTest  
mvn clean test -  
Dtest=SpringOrderRouterWithParallelMulticastTest
```

By default, the multicast will continue sending messages to destinations even if one fails. In your application, though, you may consider the whole process as failed if one destination fails. What do you do in that case?

STOPPING THE MULTICAST ON EXCEPTION

Our multicast solution at Rider Auto Parts suffers from a problem: if the order failed to send to the accounting queue, it might take longer to track down the order from production and bill the customer. To solve this problem, you can take advantage of the `stopOnException` feature of the multicast. When enabled, this feature will stop the multicast on the first exception caught, so you can take any necessary action.

To enable this feature, use the `stopOnException` method as follows:

```
from("jms:xmlOrders")  
    .multicast()  
        .stopOnException()  
        .to("direct:accounting", "direct:production")  
    .end()  
    .to("mock:end");  
  
from("direct:accounting")  
    .throwException(Exception.class, "I failed!")  
    .log("Accounting received order:  
${header.CamelFileName}")  
    .to("mock:accounting");  
  
from("direct:production")  
    .log("Production received order:  
${header.CamelFileName}")  
    .to("mock:production");
```

To handle the exception coming back from this route, you'll need

to use Camel's error-handling facilities, which are described in detail in chapter 11.

TIP Take care when using `stopOnException` with asynchronous messaging. In our example, the exception could have happened after the message had been consumed by both the accounting and production queues, nullifying the `stopOnException` effect. In our test case, we decided to use synchronous direct endpoints, which would allow us to test this feature of the multicast.

When using the XML DSL, this route looks a little different:

```
<route>
    <from uri="jms:xmlOrders"/>
    <multicast stopOnException="true">
        <to uri="direct:accounting"/>
        <to uri="direct:production"/>
    </multicast>
</route>

<route>
    <from uri="direct:accounting"/>
    <throwException exceptionType="java.lang.Exception"
message="I failed!"/>
    <log message="Accounting received order:
${header.CamelFileName}"/>
    <to uri="mock:accounting"/>
</route>
```

To run this example, go to the `chapter2/multicast` directory in the book's source code and run these commands:

```
mvn clean test -Dtest=OrderRouterWithMulticastSOETest
mvn clean test -Dtest=SpringOrderRouterWithMulticastSOETest
```

Now you know how to multicast messages in Camel, but you may be thinking that this seems like a static solution, because changing the destinations means changing the route. Let's see how to make sending to multiple recipients more dynamic.

2.6.4 USING RECIPIENT LISTS

In the previous section, you implemented a new manager's suggestion to parallelize the accounting and production queues so orders could be processed more quickly. Rider Auto Parts' top-tier customers first noticed the problem with this approach: now that all orders are going directly into production, top-tier customers aren't getting priority over the smaller customers. Their orders are taking longer, and they're losing business opportunities. Management suggested immediately going back to the old scheme, but you suggested a simple solution to the problem: by parallelizing only top-tier customers' orders, all other orders would have to go to accounting first, thereby not bogging down production.

This solution can be realized by using the Recipient List EIP. As shown in [figure 2.14](#), a recipient list first inspects the incoming message, then generates a list of desired recipients based on the message content, and sends the message to those recipients. A recipient is specified by an endpoint URI. Note that the recipient list is different from the multicast because the list of recipients is dynamic.

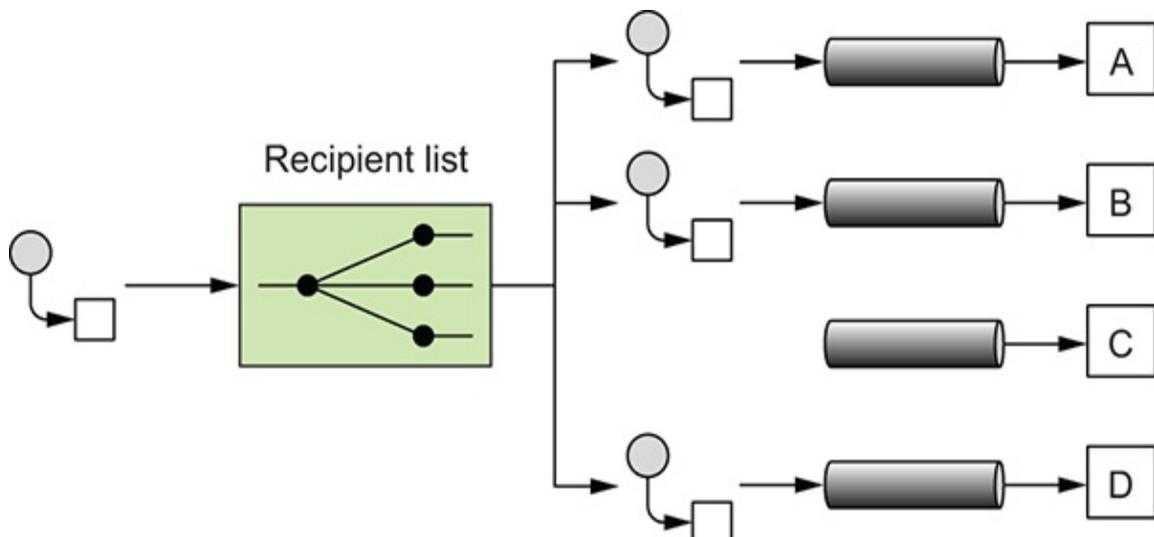


Figure 2.14 A recipient list inspects the incoming message and determines a list of recipients based on the content of the message. In this case, the message contains a list of destinations such as A, B, D. So Camel sends the message to only the A, B, and D destinations. The next message could contain a different set of destinations.

Camel provides a `recipientList` method for implementing the Recipient List EIP. For example, the following route takes the list of recipients from a header named `recipients`, where each recipient is separated from the next by a comma:

```
from("jms:xmlOrders")
    .recipientList(header("recipients"));
```

This is useful if you already have some information in the message that can be used to construct the destination names—you could use an expression to create the list. In order for the recipient list to extract meaningful endpoint URIs, the expression result must be iterable. Values that will work are `java.util.Collection`, `java.util.Iterator`, `java.util.Iterable`, Java arrays, `org.w3c.dom.NodeList`, and, as shown in the example, a string with comma-separated values.

In the Rider Auto Parts situation, the message doesn't contain that list. You need some way of determining whether the message is from a top-tier customer. A simple solution could be to call out to a custom Java bean to do this:

```
from("jms:xmlOrders")
    .setHeader("recipients", method(RecipientsBean.class,
"recipients"))
    .recipientList(header("recipients"));
```

Here `RecipientsBean` is a simple Java class as follows:

```
public class RecipientsBean {
    public String[] recipients(@XPath("/order/@customer")
String customer) {
        if (isGoldCustomer(customer)) {
            return new String[]{"jms:accounting",
"jms:production"};
        } else {
            return new String[]{"jms:accounting"};
        }
    }

    private boolean isGoldCustomer(String customer) {
        return customer.equals("honda");
    }
}
```

The `RecipientsBean` class returns "jms:accounting, jms:production" only if the customer is at the gold level of support. The check for gold-level support here is greatly simplified; ideally, you'd query a database for this check. Any other orders will be routed only to accounting, which will send them to production after the checks are complete.

The XML DSL version of this route follows a similar layout:

```
<bean id="recipientsBean"
  class="camelaction.RecipientsBean"/>

<camelContext
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="jms:xmlOrders"/>
    <setHeader headerName="recipients">
      <method ref="recipientsBean" method="recipients"/>
    </setHeader>
    <recipientList>
      <header>recipients</header>
    </recipientList>
  </route>
</camelContext>
```

The `RecipientsBean` is loaded as a Spring bean and given the name `recipientsBean`, which is then referenced in the `method` element by using the `ref` attribute.

Camel also supports a way of implementing a recipient list without using the exchange and message APIs.

RECIPIENT LIST ANNOTATION

Rather than using the `recipientList` method in the DSL, you can add a `@RecipientList` annotation to a method in a plain Java class (a Java bean). This annotation tells Camel that the annotated method should be used to generate the list of recipients from the exchange. This behavior gets invoked, however, only if the class is used with Camel's bean integration.

For example, replacing the custom bean you used in the previous section with an annotated bean results in a greatly

simplified route:

```
from("jms:xmlOrders").bean(AnnotatedRecipientList.class);
```

Now all the logic for calculating the recipients and sending out messages is captured in the `AnnotatedRecipientList` class, which looks like this:

```
public class AnnotatedRecipientList {  
    @RecipientList  
    public String[] route(@XPath("/order/@customer") String customer) {  
        if (isGoldCustomer(customer)) {  
            return new String[]{"jms:accounting",  
"jms:production"};  
        } else {  
            return new String[]{"jms:accounting"};  
        }  
    }  
  
    private boolean isGoldCustomer(String customer) {  
        return customer.equals("honda");  
    }  
}
```

Notice that the return type of the bean is a list of the desired recipients. Camel will take this list and send a copy of the message to each destination in the list.

The XML DSL version of this route follows a similar layout:

```
<bean id="annotatedRecipientList"  
      class="camelaction.AnnotatedRecipientList"/>  
  
<camelContext  
  xmlns="http://camel.apache.org/schema/spring">  
  <route>  
    <from uri="jms:xmlOrders"/>  
    <bean ref="annotatedRecipientList"/>  
  </route>  
</camelContext>
```

One nice thing about implementing the recipient list this way is that it's entirely separated from the route, which makes it a bit easier to read. You also have access to Camel's bean-binding

annotations, which allow you to extract data from the message by using expressions, so you don't have to manually explore the exchange. This example uses the @XPath bean-binding annotation to grab the customer attribute of the order element in the body. We cover these annotations in chapter 4, which is all about using beans. To run this example, go to the chapter2/recipientlist directory in the book's source code and run the command for either the Java or XML DSL case:

```
mvn clean test -  
Dtest=OrderRouterWithRecipientListAnnotationTest  
mvn clean test -  
Dtest=SpringOrderRouterWithRecipientListAnnotationTest
```

This outputs the following on the command line:

```
Accounting received order: message1.xml  
Production received order: message1.xml  
Accounting received order: message2.xml
```

Why do you get this output? Well, you had the following two orders in the src/data directory:

- message1.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<order name="motor" amount="1000" customer="honda"/>
```

- message2.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<order name="motor" amount="2" customer="joe's bikes"/>
```

The first message is from a gold customer, according to the Rider Auto Parts rules, so it was routed to both accounting and production. The second order is from a smaller customer, so it went to accounting for verification of the customer's credit standing.

What this system lacks now is a way to inspect these messages as they're flowing through the route, rather than waiting until they reach the end. Let's see how a wire tap can help.

2.6.5 USING THE WIRETAP METHOD

Often in enterprise applications, inspecting messages as they flow through a system is useful and necessary. For instance, when an order fails, you need a way to look at which messages were received to determine the cause of the failure.

You could use a simple processor, as you've done before, to output information about an incoming message to the console or append it to a file. Here's a processor that outputs the message body to the console:

```
from("jms:incomingOrders")
    .process(new Processor() {
        public void process(Exchange exchange) throws
Exception {
            System.out.println("Received order: " +
                exchange.getIn().getBody());
        }
    });
}
```

This is fine for debugging purposes, but it's a poor solution for production use. What if you wanted the message headers, exchange properties, or other data in the message exchange? Ideally, you could copy the whole incoming exchange and send that to another channel for auditing. As shown in [figure 2.15](#), the Wire Tap EIP defines such a solution.

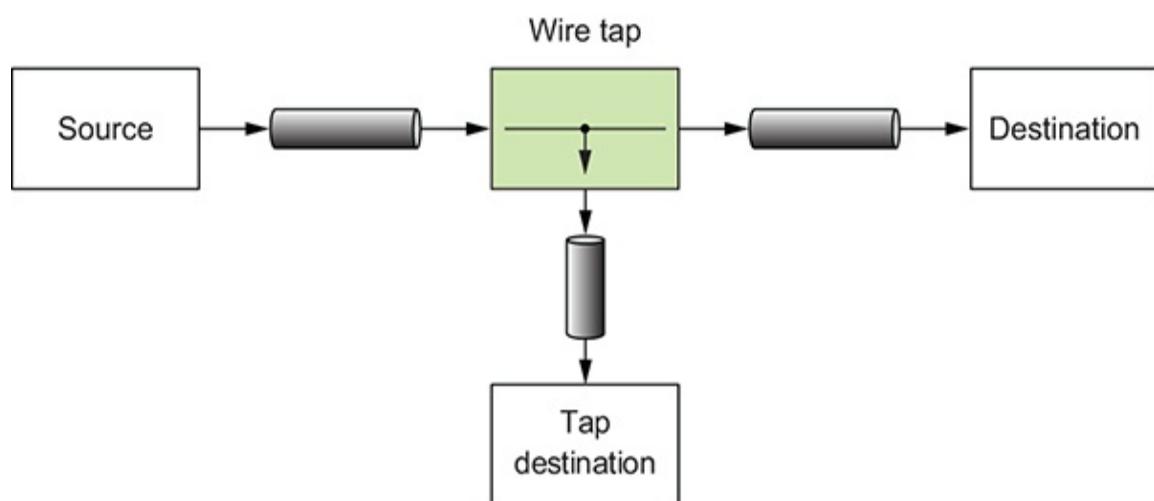


Figure 2.15 A wire tap is a fixed recipient list that sends a copy of a message traveling from a source to a destination to a secondary destination.

By using the `wireTap` method in the Java DSL, you can send a copy of the exchange to a secondary destination without affecting the behavior of the rest of the route:

```
from("jms:incomingOrders")
    .wireTap("jms:orderAudit")
    .choice()
        .when(header("CamelFileName").endsWith(".xml"))
            .to("jms:xmlOrders")
        .when(header("CamelFileName").regex("^.*" +
(csv|csl)$"))
            .to("jms:csvOrders")
        .otherwise()
            .to("jms:badOrders");
```

The preceding code sends a copy of the exchange to the `orderAudit` queue, and the original exchange continues on through the route, as if you hadn't used a wire tap at all. Camel doesn't wait for a response from the wire tap because the wire tap sets the message exchange pattern (MEP) to `InOnly`. The message will be sent to the `orderAudit` queue in a fire-and-forget fashion—it won't wait for a reply.

In the XML DSL, you can configure a wire tap just as easily:

```
<route>
    <from uri="jms:incomingOrders"/>
    <wireTap uri="jms:orderAudit"/>
    ...
```

To run this example, go to the `chapter2/wiretap` directory in the book's source code and run these commands:

```
mvn clean test -Dtest=OrderRouterWithWireTapTest
mvn clean test -Dtest=SpringOrderRouterWithWireTapTest
```

What can you do with a tapped message? Numerous things could be done at this point:

- You could print the information to the console as you did before. This is useful for simple debugging purposes.
- You could save the message in a persistent store (in a file or database) for retrieval later.

The wire tap is a useful monitoring tool, but it leaves most of the work up to you. We'll discuss some of Camel's more powerful tracing and auditing tools in chapter 16.

2.7 Summary and best practices

In this chapter, we've covered probably the most prominent ability of Camel: routing messages. By now you should know how to create routes in either the Java or XML DSL and know the differences in their configuration. You should also have a good grasp of when to apply several EIP implementations in Camel and how to use them. With this knowledge, you can create Camel applications that do useful tasks.

Here are some of the key concepts you should take away from this chapter:

- *Routing occurs in many aspects of everyday life*—Whether you're surfing the internet, doing online banking, or booking a flight or hotel room, messages are being routed behind the scenes via some sort of router.
- *Use Apache Camel for routing messages*—Camel is primarily a message router that allows you to route messages from and to a variety of transports and APIs.
- *Camel's DSLs are used to define routing rules*—The Java DSL allows you to write in the popular Java language, which gives you autocompletion of terms in most IDEs. It also allows you to use the full power of the Java language when writing routes. It's considered the main DSL in Camel. The XML DSL allows you to write routing rules without any Java code at all.
- *The Java DSL and Spring camelContext are a powerful combination*—Section 2.4.3 described our favorite way to write Camel applications, which is to boot up `camelContext` in Spring and write routing rules in Java DSL `RouteBuilders`. This gives you the best of both: the most expressive DSL that Camel has in the Java DSL, and a more feature-rich and standard container in

the Spring CamelContext.

- *Use enterprise integration patterns (EIPs) to solve integration and routing problems*—EIPs are like design patterns from object-oriented programming, but for the enterprise integration world.
- *Use Camel's built-in EIP implementations rather than creating your own*—Camel implements most EIPs as easy-to-use DSL terms, which allows you to focus on the business problem rather than the integration architecture.

The coming chapters build on this foundation to show you things like data transformation, using beans, using more advanced EIPs, sending data over other transports, and more. In the next chapter, you'll look at how Camel makes data transformation a breeze.

Part 2

Core Camel

In part 1, we guided you through what we consider introductory topics in Camel—topics you absolutely need to know to use Camel. In this part, we'll cover the core features of Camel in depth. You'll need many of these features when using Camel in real-world applications.

In chapter 3, we'll take a look at the data in the messages being routed by Camel. In particular, you'll see how to transform this data to other formats by using Camel.

Camel has great support for integrating beans into your routing applications. In chapter 4, we'll look at the many ways of using beans in Camel applications. Chapter 5 of this part revisits the important topic of enterprise integration patterns (EIPs) in Camel. Back in chapter 2, we covered some of the simpler EIPs; in chapter 5, we'll look at several of the more complex EIPs.

Components are the main extension mechanism in Camel. As such, they include functionality to connect to many different transports, APIs, and other extensions to Camel's core. Chapter 6 covers the most heavily used components that ship with Camel

3

Transforming data with Camel

This chapter covers

- Transforming data by using EIPs and Java
- Transforming XML data
- Transforming by using well-known data formats
- Writing your own data formats for transformations
- Understanding the Camel type-converter mechanism

The preceding chapter covered routing, which is the single most important feature any integration kit must provide. This chapter looks at the second most important feature: data or message transformation.

Just like the real world, where people speak different languages, the IT world speaks different protocols. Software engineers regularly need to act as mediators between various protocols when IT systems must be integrated. To address this, the data models used by the protocols must be transformed from one form to another, adapting to whatever protocol the receiver understands. Mediation and data transformation are key features in any integration kit, including Camel.

In this chapter, you'll learn all about how Camel can help you with your data transformation challenges. We'll start with a brief overview of data transformation in Camel and then look at

transforming data into any custom format you may have. Next we'll look at Camel components that are specialized for transforming XML data and other well-known data formats. We end the chapter by looking into Camel's type-converter mechanism, which supports, implicitly and explicitly, type conversion.

After reading this chapter, you'll know how to tackle any data transformation you're faced with and which Camel solution to use.

3.1 Data transformation overview

Camel provides many techniques for data transformation, and we'll cover them shortly. But let's start with an overview of data transformation in Camel. *Data transformation* is a broad term that covers two types of transformation:

- *Data format transformation*—The data format of the message body is transformed from one form to another. For example, a CSV record is formatted as XML.
- *Data type transformation*—The data type of the message body is transformed from one type to another. For example, `java.lang.String` is transformed into `javax.jms.TextMessage`.

Figure 3.1 illustrates the principle of transforming a message body from one form into another. This transformation can involve any combination of format and type transformations. In most cases, the data transformation you'll face with Camel is format transformation: you have to mediate between two protocols. Camel has a built-in type-converter mechanism that can automatically convert between types, which greatly reduces the need for end users to deal with type transformations.

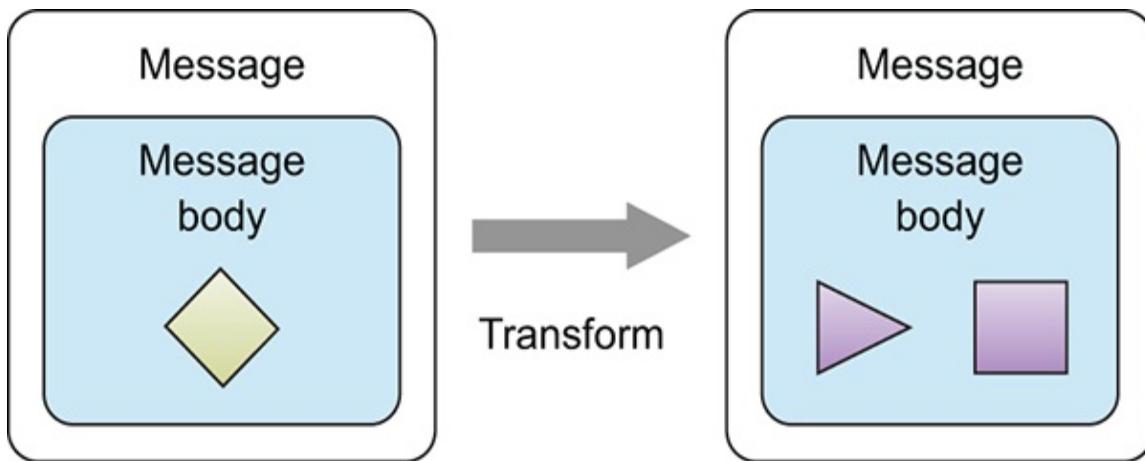


Figure 3.1 Camel offers many features for transforming data from one form to another.

Camel has many data-transformation features. We introduce them in the following section, and then present them one by one. After reading this chapter, you'll have a solid understanding of how to use Camel to transform your data. In Camel, data transformation typically takes place in the six ways listed in table 3.1.

Table 3.1 Six ways data transformation typically takes place in Camel

Transformation	Description
Data transformation using EIPs and Java	You can explicitly enforce transformation in the route by using the Message Translator or the Content Enricher EIPs. This gives you the power to do data mapping by using regular Java code. We cover this in section 3.2.
Data transformation using components	Camel provides a range of components for transformation, such as the XSLT component for XML transformation. We dive into this in section 3.3.
Data transformation using data	Data formats are Camel transformers that come in pairs to transform data back and forth between well-known formats. Section 3.4 covers this topic.

formats	
Data transformation using templates	Camel provides a range of components for transforming by using templates, such as Apache Velocity. We'll look at this in section 3.5.
Data type transformation using Camel's type-converter mechanism	Camel has an elaborate type-converter mechanism that activates on demand. This is convenient when you need to convert from common types such as <code>java.lang.Integer</code> to <code>java.lang.String</code> or even from <code>java.io.File</code> to <code>java.lang.String</code> . Section 3.6 covers type converters.
Message transformation in component adapters	Camel's many components adapt to various commonly used protocols and, as such, need to be able to transform messages as they travel to and from those protocols. Often these components use a combination of custom data transformations and type converters. This happens seamlessly, and only component writers need to worry about it. Chapter 8 covers writing custom components.

This chapter covers the first five of these data transformation methods. We'll leave the last one for chapter 8 because it applies only to writing custom components.

3.2 Transforming data by using EIPs and Java

Data mapping, the process of mapping between two distinct data models, is a key factor in data integration. There are many existing standards for data models, governed by various organizations or committees. As such, you'll often find yourself needing to map from a company's custom data model to a standard data model.

Camel provides great freedom in data mapping because it allows you to use Java code. You aren't limited to using a particular data-mapping tool that may at first seem elegant but turns out to make things impossible.

In this section, you'll look at mapping data by using Processor, a Camel API. Camel can also use Java beans for mapping, which is a good practice because it allows your mapping logic to be independent of the Camel API.

3.2.1 USING THE MESSAGE TRANSLATOR EIP

The Message Translator EIP is illustrated in [figure 3.2](#).

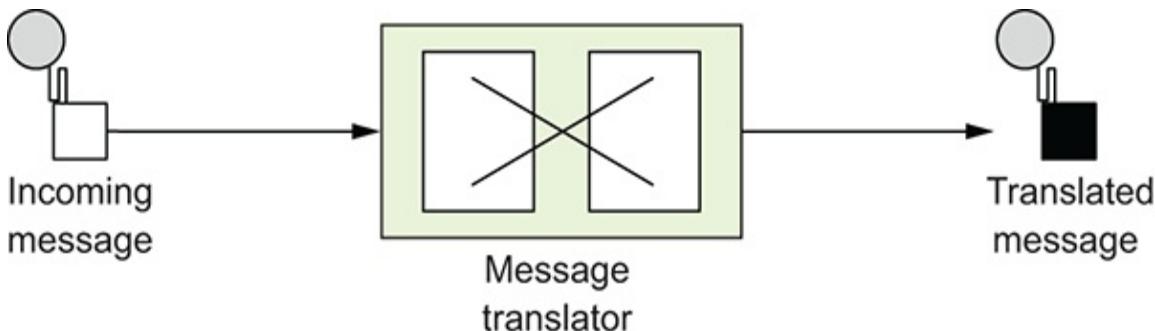


Figure 3.2 In the Message Translator EIP, an incoming message goes through a translator and comes out as a translated message.

This pattern covers translating a message from one format to another. It's the equivalent of the Adapter pattern from the Gang of Four book.

NOTE The Gang of Four book is *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994). See the “Design Patterns” Wikipedia article for more information:

[http://en.wikipedia.org/wiki/Design_Patterns_\(book\)](http://en.wikipedia.org/wiki/Design_Patterns_(book)).

Camel provides three ways of using this pattern:

- Using Processor
- Using Java beans
- Using <transform>

We'll look at them each in turn.

TRANSFORMING USING PROCESSOR

The Camel Processor is an interface defined in `org.apache.camel.Processor` with a single method:

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

Processor is a low-level API in which you work directly on the Camel Exchange instance. It gives you full access to all of Camel's moving parts from the `camelContext`, which you can obtain from the Exchange by using the `getContext` method.

Let's look at an example. At Rider Auto Parts, you've been asked to generate daily reports of newly received orders to be outputted to a CSV file. The company uses a custom format for order entries, but to make things easy, they already have an HTTP service that returns a list of orders for whatever date you input. The challenge you face is mapping the returned data from the HTTP service to a CSV format and writing the report to a file.

Because you want to get started on a prototype quickly, you decide to use the Camel Processor, as shown in the following listing.

Listing 3.1 Using Processor to translate from a custom format to a CSV format

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

public class OrderToCsvProcessor implements Processor {
    public void process(Exchange exchange) throws Exception
    {
        String custom = exchange.getIn()
            .getBody(String.class); 1
    }
}
```

1

Gets custom payload

```
String id = custom.substring(0, 10); 2
```

2

Extracts data to local variables

```
String customerId = custom.substring(10, 20); ②  
String date = custom.substring(20, 30); ②  
String items = custom.substring(30); ②  
String[] itemIds = items.split("@"); ②  
StringBuilder csv = new StringBuilder(); ③
```

3

Maps to CSV format

```
csv.append(id.trim()); ③  
csv.append(",").append(date.trim()); ③  
csv.append(",").append(customerId.trim()); ③  
for (String item : itemIds) { ③  
    csv.append(",").append(item.trim()); ③  
}  
③ exchange.getIn().setBody(csv.toString()); ④
```

4

Replaces payload with CSV payload

```
}
```

First you grab the custom format payload from the exchange ①. It's a String type, so you pass string in as the parameter to have the payload returned as a string. Then you extract data from the custom format to the local variables ②. The custom format could be anything, but in this example, it's a fixed-length custom format. Then you map the CSV format by building a string with comma-separated values ③. Finally, you replace the custom payload with your new CSV payload ④.

You can use `OrderToCsvProcessor` from [listing 3.1](#) in a Camel route as follows:

```
from("quartz2://report?cron=0+0+6+*+*+?")  
.to("http://riders.com/orders/cmd=received&date=yesterday")
```

```
.process(new OrderToCsvProcessor())
.to("file://riders/orders?fileName=report-
${header.Date}.csv");
```

The preceding route uses Quartz to schedule a job to run once a day at 6 a.m. It then invokes the HTTP service to retrieve the orders received yesterday, which are returned in the custom format. Next, it uses `OrderToCsvProcessor` to map from the custom format to CSV format before writing the result to a file.

The equivalent route in XML is as follows:

```
<bean id="csvProcessor"
class="camelaction.OrderToCsvProcessor"/>

<camelContext
xmlns="http://camel.apache.org/schema/spring">
<route>
<from uri="quartz2://report?cron=0+0+6+*+*+?" />
<to
uri="http://riders.com/orders/cmd=received&date=yesterd
ay"/>
<process ref="csvProcessor"/>
<to uri="file://riders/orders?fileName=report-
${header.Date}.csv"/>
</route>
</camelContext>
```

You can try this example yourself; we've provided a little unit test with the book's source code. Go to the `chapter3/transform` directory and run these Maven goals:

```
mvn test -Dtest=OrderToCsvProcessorTest
mvn test -Dtest=SpringOrderToCsvProcessorTest
```

After the test runs, a report file is written in the `target/orders/received` directory.

Using the `getIn` and `getOut` methods on exchanges

The Camel Exchange defines two methods for retrieving

messages: `getIn` and `getOut`. The `getIn` method returns the incoming message, and the `getOut` method accesses the outbound message.

In two scenarios, the Camel end user will have to decide which method to use:

- A read-only scenario, such as when you're logging the incoming message
- A write scenario, such as when you're transforming the message

In the second scenario, you'd assume `getOut` should be used. That's correct according to theory, but in practice there's a common pitfall when using `getOut`: the incoming message headers and attachments will be lost. This is often not what you want, so you must copy the headers and attachments from the incoming message to the outgoing message, which can be tedious. The alternative is to set the changes directly on the incoming message by using `getIn`, and not to use `getOut` at all. This is the practice we use most often in this book.

Using a processor has one disadvantage: you're required to use the Camel API. In the next section, you'll learn how to avoid this by using a bean.

TRANSFORMING USING BEANS

Using beans is a great practice because it allows you to use any Java code and library you wish. Camel imposes no restrictions whatsoever. Camel can invoke any bean you choose, so you can use existing beans without having to rewrite or recompile them.

The following listing shows using a bean instead of Processor.

Listing 3.2 Using a bean to translate from a custom format to CSV format

```
public class OrderToCsvBean {  
    public static String map(String custom) {  
        String id = custom.substring(0, 10); 1  
    }  
}
```

1

Extracts data to local variables

```
String customerId = custom.substring(10, 20); 1  
String date = custom.substring(20, 30); 1  
String items = custom.substring(30); 1  
String[] itemIds = items.split("@"); 1  
StringBuilder csv = new StringBuilder(); 1  
csv.append(id.trim());  
csv.append(",").append(date.trim());  
csv.append(",").append(customerId.trim());  
for (String item : itemIds) {  
    csv.append(",").append(item.trim());  
}  
return csv.toString(); 2
```

2

Returns CSV payload

```
}  
}
```

The first noticeable difference between listings [3.1](#) and [3.2](#) is that listing [3.2](#) doesn't use any Camel imports. Your bean is totally independent of the Camel API. The next difference is that you can name the method signature in [listing 3.2](#)—in this case, it's a static method named `map`.

The method signature defines the contract, which means that the first parameter (`String custom`) is the message body you're going to use for translation. The method returns a string, which means the translated data will be a `String` type. At runtime, Camel binds to this method signature. We won't go into any more details here; chapter 4 covers much more about using beans.

The mapping **1** is the same as with the processor. At the end, you return the mapping output **2**.

You can use `OrderToCsvBean` in a Camel route as shown here:

```
from("quartz2://report?cron=0+0+6+*+*+?")  
    .to("http://riders.com/orders/cmd=received&date=yesterday")  
        .bean(new OrderToCsvBean())  
        .to("file://riders/orders?fileName=report-  
${header.Date}.csv");
```

The equivalent route in XML is as follows:

```
<bean id="csvBean" class="camelaction.OrderToCsvBean"/>  
  
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
    <route>  
        <from uri="quartz2://report?cron=0+0+6+*+*+?"/>  
        <to  
            uri="http://riders.com/orders/cmd=received&date=yesterd  
ay"/>  
            <bean ref="csvBean"/>  
            <to uri="file://riders/orders?fileName=report-  
${header.Date}.csv"/>  
    </route>  
</camelContext>
```

You can try this example from the `chapter3/transform` directory by using the following Maven goals:

```
mvn test -Dtest=OrderToCsvBeanTest  
mvn test -Dtest=SpringOrderToCsvBeanTest
```

This generates a test report file in the `target/orders/received` directory.

Another advantage of using beans over processors for mappings is that unit testing is much easier. For example, [listing 3.2](#) doesn't require the use of Camel at all, as opposed to [listing 3.1](#), where you need to create and pass in an Exchange instance.

We'll leave the beans for now, because they're covered extensively in the next chapter. But you should keep in mind that beans are useful for doing message transformation.

TRANSFORMING USING THE TRANSFORM METHOD FROM THE JAVA DSL

`transform` is a method in the Java DSL that can be used in Camel routes to transform messages. By allowing the use of expressions, `transform` permits great flexibility, and using expressions directly within the DSL can sometimes save time. Let's look at a little example.

Suppose you need to prepare text for HTML formatting by replacing all line breaks with a `
` tag. You can do this with a built-in Camel expression that searches and replaces using regular expressions:

```
from("direct:start")
    .transform(body().regexReplaceAll("\n", "<br/>"))
    .to("mock:result");
```

What this route does is use the `transform` method to tell Camel that the message should be transformed using an expression. Camel provides the Builder pattern to build compound expressions from individual expressions. This is done by chaining together method calls, which is the essence of the Builder pattern.

NOTE For more information on the Builder pattern, see the Wikipedia article: http://en.wikipedia.org/wiki/Builder_pattern.

In this example, you combine `body` and `regexReplaceAll`. The expression should be read as follows: take the `body` and perform a regular expression that replaces all new lines (`\n`) with `
` tags. Now you've combined two methods that conform to a compound Camel expression.

You can run this example from `chapter3/transform` directly by using the following Maven goal:

```
mvn test -Dtest=TransformTest
```

The Direct component

The example here uses the Direct component (<http://camel.apache.org/direct>) as the input source for the route (from("direct:start")). The Direct component provides direct invocation between a producer and a consumer. It allows connectivity only from within Camel, so external systems can't send messages directly to it. This component is used within Camel to do things such as link routes together or for testing.

For more information on the Direct component and other types of in-memory messaging, see chapter 6.

Camel also allows you to use custom expressions. This is useful when you need to be in full control and have Java code at your fingertips. For example, the previous example could've been implemented as follows:

```
from("direct:start")
    .transform(new Expression() {
        public <T> T evaluate(Exchange exchange, Class<T>
type) {
            String body =
exchange.getIn().getBody(String.class);
            body = body.replaceAll("\n", "<br/>");
            body = "<body>" + body + "</body>";
            return (T) body;
        }
    })
    .to("mock:result");
```

As you can see, this code uses an inlined Camel Expression that allows you to use Java code in its evaluate method. This follows the same principle as the Camel Processor you saw before.

Now let's see how to transform data using the XML DSL.

TRANSFORMING USING <TRANSFORM> FROM THE XML DSL

Using <transform> from the XML DSL is a bit different from the Java DSL because the XML DSL isn't as powerful. In the XML

DSL, the Builder pattern expressions aren't available because with XML you don't have a real programming language underneath. What you can do instead is invoke a method on a bean or use scripting languages.

Let's see how this works. The following route uses a method call on a bean as the expression:

```
<bean id="htmlBean" class="camelaction.HtmlBean"/> 1
```

1

Does the transformation

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
<route>
    <from uri="direct:start"/>
    <transform>
        <method bean="htmlBean" method="toHtml"/> 2
    </transform>
    <to uri="mock:result"/>
</route>
</camelContext>
```

2

Invokes toHtml method on bean

```
</transform>
<to uri="mock:result"/>
</route>
</camelContext>
```

First, you declare a regular Spring bean to be used to transform the message **1**. Then, in the route, you use `<transform>` with a `<method>` call expression to invoke the bean **2**.

The implementation of the `htmlBean` is straightforward:

```
public class HtmlBean {
    public static String toHtml(String body) {
        body = body.replaceAll("\n", "<br/>");
        body = "<body>" + body + "</body>";
        return body;
    }
}
```

You can also use scripting languages as expressions in Camel. For example, you can use Groovy, MVFLEX Expression Language (MVEL), JavaScript, or Camel's own scripting language, called Simple (explained in appendix A). We won't go into detail on how to use the other scripting languages at this point, but you can use the Simple language to build strings with placeholders. It pretty much speaks for itself—we're sure you'll understand what the following transformation does:

```
<transform>
    <simple>Hello ${body} how are you?</simple>
</transform>
```

You can try the XML DSL transformation examples provided in the book's source code by running the following Maven goals from the chapter3/transform directory:

```
mvn test -Dtest=SpringTransformMethodTest
mvn test -Dtest=SpringTransformScriptTest
```

We're done covering the Message Translator EIP, so let's look at the related Content Enricher EIP.

3.2.2 USING THE CONTENT ENRICHER EIP

The Content Enricher EIP is illustrated in [figure 3.3](#). This pattern documents the scenario in which a message is enriched with data obtained from another resource.

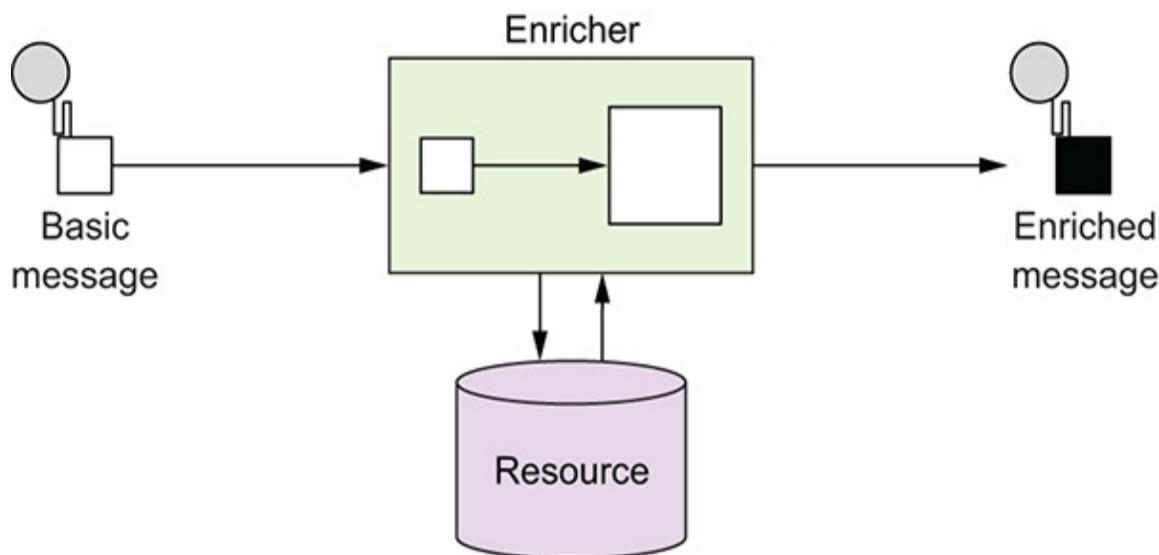


Figure 3.3 In the Content Enricher EIP, an existing message has data added to it from another source.

To help understand this pattern, let's turn back to Rider Auto Parts. It turns out that the data mapping you did in [listing 3.1](#) wasn't sufficient. Orders are also piled up on an FTP server, and your job is to somehow merge this information into the existing report. [Figure 3.4](#) illustrates the scenario.

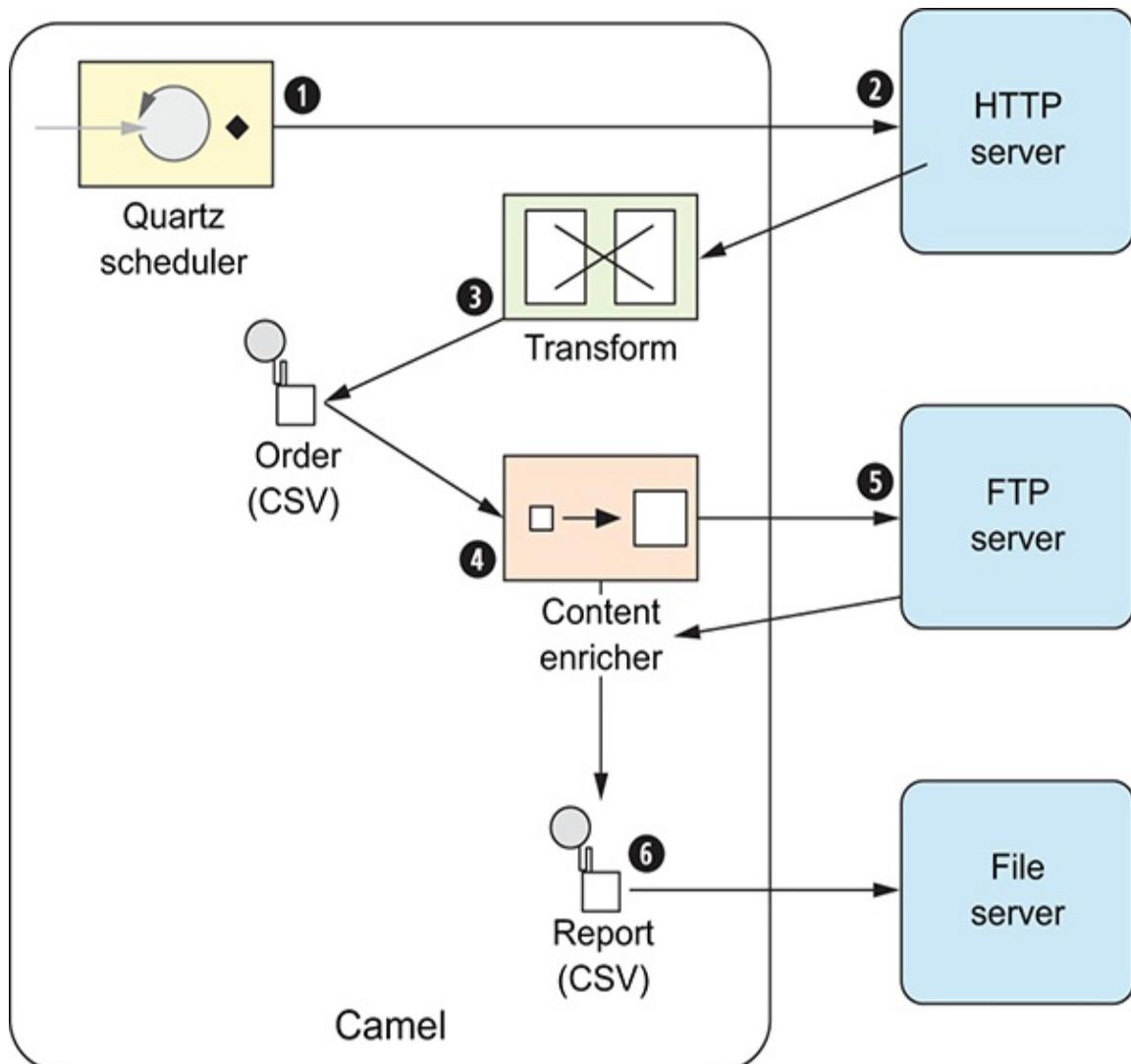


Figure 3.4 An overview of the route that generates the orders report, now with the content enricher pulling in data from an FTP server

A scheduled consumer using Quartz starts the route every day at 6 a.m. (1). It then pulls data from an HTTP server, which returns orders in a custom format (2), which is then transformed into

CSV format ③. At this point, you have to perform the additional content enrichment step ④ with the data obtained from the FTP server ⑤. After this, the final report is written to the file server ⑥.

Before you dig into the code and see how to implement this, you need to take a step back and look at how the Content Enricher EIP is implemented in Camel. Camel provides two methods in the DSL for implementing the pattern:

- `pollEnrich`—This method merges data retrieved from another source by using a consumer.
- `enrich`—This method merges data retrieved from another source by using a producer.

The difference between `pollEnrich` and `enrich`

The difference between `pollEnrich` and `enrich` is that the former uses a consumer, and the latter uses a producer, to retrieve data from the source. Knowing the difference is important: the file component can be used with both, but using `enrich` will write the message content as a file; using `pollEnrich` will read the file as the source, which is most likely the scenario you'll be facing when enriching with files. The HTTP component works only with `enrich`; it allows you to invoke an external HTTP service and use its reply as the source.

Camel uses the `org.apache.camel.processor.aggregate.AggregationStrategy` interface to merge the result from the source with the original message, as follows:

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

This aggregate method is a callback that you must implement. The method has two parameters: the first, named `oldExchange`, contains the original exchange; the second, `newExchange`, is the enriched source. Your task is to enrich the message by using Java code and return the merged result. Let's see this in action.

To solve the problem at Rider Auto Parts, you need to use `pollEnrich` because it's capable of polling a file from an FTP server.

ENRICHING USING POLLENRICH

The following listing shows how to use `pollEnrich` to retrieve the additional orders from the remote FTP server and aggregate this data with the existing message by using Camel's `AggregationStrategy`.

Listing 3.3 Using `pollEnrich` to merge additional data with an existing message

```
from("quartz2://report?cron=0+0+6+*+*+?")  
    .to("http://riders.com/orders/cmd=received&date=yesterday")  
    .process(new OrderToCsvProcessor())  
    .pollEnrich("ftp://riders.com/orders/?  
username=rider&password=secret",  
            new AggregationStrategy() { ①
```

①

Uses `pollEnrich` to read FTP file

```
        public Exchange aggregate(Exchange oldExchange,  
                                Exchange  
newExchange) {  
            if (newExchange == null) {  
                return oldExchange;  
            }  
            String http = oldExchange.getIn() ②
```

②

Merges data using `AggregationStrategy`

```

        .getBody(String.class);      ②
    String ftp = newExchange.getIn()      ②
        .getBody(String.class);      ②
    String body = http + "\n" + ftp;      ②
    oldExchange.getIn().setBody(body);      ②
    return oldExchange;
}
})
.to("file://riders/orders");          ③

```

③

Writes output to file

The route is triggered by Quartz to run at 6 a.m. every day. You invoke the HTTP service to retrieve the orders and transform them to CSV format by using a processor.

At this point, you need to enrich the existing data with the orders from the remote FTP server. This is done by using `pollEnrich` ①, which consumes the remote file.

To merge the data, you use `AggregationStrategy` ②. First, you check whether any data was consumed. If `newExchange` is `null`, there's no remote file to consume, and you just return the existing data. If there's a remote file, you merge the data by concatenating the existing data with the new data and setting it back on the `oldExchange`. Then, you return the merged data by returning the `oldExchange`. To write the CSV report file, you use the `file` component ③.

TIP Both `enrich` and `pollEnrich` can accept dynamic URIs, as discussed in chapter 2, section 2.5.1.

`PollEnrich` uses a polling consumer to retrieve messages, and it offers three time-out modes:

- `pollEnrich(timeout=-1)`—Polls the message and waits until a message arrives. This mode blocks until a message exists.

- `pollEnrich(timeout = 0)`—Immediately polls the message if any exists; otherwise, `null` is returned. It never waits for messages to arrive, so this mode never blocks. This is the default mode.
- `pollEnrich(timeout>0)`—Polls the message, and if no message exists, it waits for one, waiting at most until the time-out triggers. This mode potentially blocks.

It's a best practice to either use `timeout = 0` or assign a time-out value when using `pollEnrich` to avoid waiting indefinitely if no message arrives.

Now let's take a quick look at how to use `enrich` with the XML DSL; it's a bit different from using the Java DSL. You use `enrich` when you need to enrich the current message with data from another source using request-reply messaging. A prime example is to enrich the current message with the reply from a web service call. But let's look at another example, using XML to enrich the current message via the TCP transport:

```
<bean id="quoteStrategy"
      class="camelaction.QuoteStrategy"/>
```

①

①

Bean implementing AggregationStrategy

```
<route>
    <from uri="jms:queue:quotes"/>
    <enrich url="netty4:tcp://riders.com:9876?
textline=true&sync=true"
            strategyRef="quoteStrategy"/>
    <to uri="log:quotes"/>
</route>
```

Here you use the Camel `netty4` component for the TCP transport, configured to use request-reply messaging by using the `sync=true` option. To merge the original message with data from the remote server, `<enrich>` must refer to an `AggregationStrategy`. This is done using the `strategyRef` attribute. As you can see in the example, the `quoteStrategy` being referred to is a bean id ①, which contains the implementation of

the `AggregationStrategy`, where the merging takes place.

You've seen a lot about how to transform data in Camel, using Java code for the transformations. Now let's take a peek into the XML world and look at the XSLT component, which is used for transforming XML messages into another format by using XSLT stylesheets.

3.3 Transforming XML

Camel provides two ways to perform XML transformations:

- *XSLT component*—For transforming an XML payload into another format by using XSLT stylesheets
- *XML marshaling*—For marshaling and unmarshaling objects to and from XML

Both of these are covered in the following subsections.

3.3.1 TRANSFORMING XML WITH XSLT

XSL Transformations (XSLT) is a declarative XML-based language used to transform XML documents into other documents. For example, XSLT can be used to transform XML into HTML for web pages or to transform an XML document into another XML document with a different structure. XSLT is powerful and versatile, but it's also a complex language that takes time and effort to fully understand and master. Think twice before deciding to pick up and use XSLT.

Camel provides the XSLT component as part of `camel-core.jar`, so you don't need any other dependencies. Using the XSLT component is straightforward because it's just another Camel component. The following route shows an example of how to use it; this route is also illustrated in [figure 3.5](#):

```
from("file:///rider/inbox")
    .to("xslt://camelinaction/transform.xsl")
    .to("jms:queue:transformed")
```

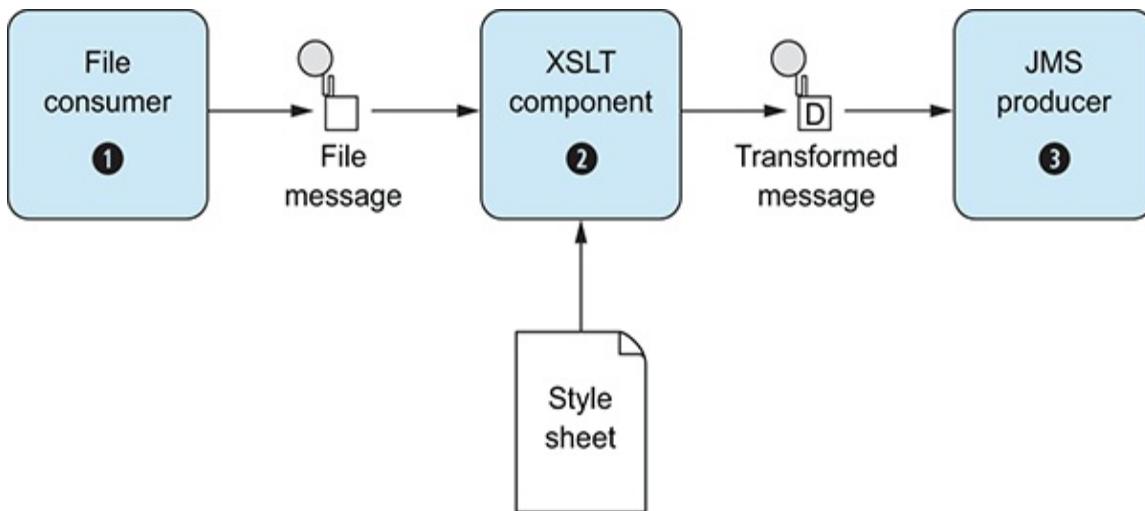


Figure 3.5 A Camel route using an XSLT component to transform an XML document before it's sent to a JMS queue

The file consumer picks up new files and routes them to the XSLT component, which transforms the payload by using the stylesheet. After the transformation, the message is routed to a JMS producer, which sends the message to the JMS queue. Notice in the preceding code how the URI for the XSLT component is defined: `xslt://camelaction/transform.xsl`. The part after the scheme is the URI location of the stylesheet to use. Camel will look in the classpath by default. To look elsewhere, you can prefix the resource name with any of the prefixes listed in table 3.2.

Table 3.2 Prefixes supported by the XSLT component for loading stylesheets

Prefix	Example	Description
<n on e>	<code>xslt://camelaction/transform.xsl</code>	If no prefix is provided, Camel loads the resource from the classpath.
fi le :	<code>xslt://file:/rider/config/transform.xml</code>	Loads the resource from the filesystem.
ht tp :	<code>xslt://http://rider.com/styles/transform.xsl</code>	Loads the resource from a URL.
re	<code>xslt://ref:resourceId</code>	Look up the resource from the registry.

f:		
be an :	xslt://bean:nameOfBean.met hodName	Look up a bean in the registry and call a method which returns the resource.

Let's leave the XSLT world now and take a look at how to do XML-to-object marshaling with Camel.

3.3.2 TRANSFORMING XML WITH OBJECT MARSHALING

Any software engineer who has worked with XML knows that it's a challenge to use the low-level XML API that Java offers.

Instead, people often prefer to work with regular Java objects and use marshaling to transform between Java objects and XML representations.

In Camel, this marshaling process is provided in ready-to-use components known as *data formats*. Section 3.4 covers data formats in full detail, but you'll take a quick look at the XStream and JAXB data formats here as we cover XML transformations using marshaling.

TRANSFORMING USING XSTREAM

XStream is a simple library for serializing objects to XML and back again. To use it, you need camel-xstream.jar on the classpath and the XStream library itself.

Suppose you need to send messages in XML format to a shared JMS queue, which is then used to integrate two systems. The following listing shows how this can be done.

Listing 3.4 Using XStream to transform a message into XML

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
        <xstream id="myXstream"/>
```

①

1

Specifies XStream data format

```
</dataFormats>
<route>
    <from uri="direct:foo"/>
    <marshal ref="myXstream"/>
```

2

2

Transforms to XML

```
<to uri="jms:queue:foo"/>
</route>
</camelContext>
```

When using the XML DSL, you can declare the data formats used at the top 1 of the `<camelContext>`. By doing this, you can share the data formats in multiple routes. In the first route, where you send messages to a JMS queue, you use `marshal` 2, which refers to the `id` from 1, so Camel knows that the XStream data format is being used.

You can also use the XStream data format directly in the route, which can shorten the syntax a bit, like this:

```
<route>
    <from uri="direct:foo"/>
    <marshal><xstream/></marshal>
    <to uri="jms:queue:foo"/>
</route>
```

The same route is shorter to write in the Java DSL, because you can do it with one line per route:

```
from("direct:foo").marshal().xstream().to("jms:queue:foo");
```

Yes, using XStream is that simple. And the reverse operation, unmarshaling from XML to an object, is just as simple:

```
<route>
    <from uri="jms:queue:foo"/>
    <unmarshal ref="myXstream"/>
```

```
<to uri="direct:handleFoo"/>  
</route>
```

You've now seen how easy it is to use XStream with Camel. Let's take a look at using JAXB with Camel.

TRANSFORMING USING JAXB

Java Architecture for XML Binding (JAXB) is a standard specification for XML binding, and it's provided out of the box in the Java runtime. Like XStream, it allows you to serialize objects to XML and back again. It's not as simple, but it does offer more bells and whistles for controlling the XML output. And because it's distributed in Java, you don't need any special JAR files on the classpath.

Unlike XStream, JAXB requires that you do a bit of work to declare the binding between Java objects and the XML form. This is done using annotations. Suppose you define a model bean to represent an order, as shown in [listing 3.5](#), and you want to transform this into XML before sending it to a JMS queue. Then you want to transform it back to the order bean again when consuming from the JMS queue. This can be done as shown in [listings 3.5](#) and [3.6](#).

Listing 3.5 Annotating a bean with JAXB so it can be transformed to and from XML

```
package camelinaction;  
  
import javax.xml.bind.annotation.XmlAccessType;  
import javax.xml.bind.annotation.XmlAccessorType;  
import javax.xml.bind.annotation.XmlAttribute;  
import javax.xml.bind.annotation.XmlRootElement;  
  
@XmlRootElement ①
```

①

PurchaseOrder class is JAXB annotated

```
@XmlAccessorType(XmlAccessType.FIELD) ①
public class PurchaseOrder { ①
    @XmlAttribute
    private String name;
    @XmlAttribute
    private double price;
    @XmlAttribute
    private double amount;
}
```

Listing 3.5 shows how to use JAXB annotations to decorate your model object (omitting the usual getters and setters). First you define `@XmlRootElement` ① as a class-level annotation to indicate that this class is an XML element. Then you define the `@XmlAccessorType` to let JAXB access fields directly. To expose the fields of this model object as XML attributes, you mark them with the `@XmlAttribute` annotation.

Using JAXB, you should be able to marshal a model object into an XML representation like this:

```
<purchaseOrder name="Camel in Action" price="6999"
amount="1"/>
```

The following listing shows how you can use JAXB in routes to transform the `PurchaseOrder` object to XML before it's sent to a JMS queue, and then back again from XML to the `PurchaseOrder` object when consuming from the same JMS queue.

Listing 3.6 Using JAXB to serialize objects to and from XML

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
        <jaxb id="jaxb" contextPath="camelinaction"/> ①
    </dataFormats>
    <route>
        <from uri="direct:order"/>
        <transform>
            <marshal type="jaxb"/>
        </transform>
        <to uri="jms:queue:purchaseorders"/>
    </route>
</camelContext>
```

①

Declares JAXB data format

```
</dataFormats>
<route>
    <from uri="direct:order"/>
```

```
<marshal ref="jaxb"/> 2
```

2

Transforms from model to XML

```
<to uri="jms:queue:order"/>
</route>
<route>
    <from uri="jms:queue:order"/>
    <unmarshal ref="jaxb"/> 3
```

3

Transforms from XML to model

```
<to uri="direct:doSomething"/>
</route>
</camelContext>
```

First you need to declare the JAXB data format **1**. Note that a contextPath attribute is also defined on the JAXB data format; this is a package name that instructs JAXB to look in this package for classes that are JAXB annotated. The first route then marshals to XML **2**, and the second route unmarshals to transform the XML back into the PurchaseOrder object **3**.

You can try this example by running the following Maven goal from the chapter3/order directory:

```
mvn test -Dtest=PurchaseOrderJaxbTest
```

NOTE To tell JAXB which classes are JAXB annotated, you need to drop a special jaxb.index file into each package in the classpath containing the POJO classes. It's a plain-text file in which each line lists the class name. In the preceding example, the file contains a single line with the text PurchaseOrder.

That's the basis of using XML object marshaling with XStream and JAXB. Both are implemented in Camel via data formats that

are capable of transforming back and forth between various well-known formats.

3.4 Transforming with data formats

In Camel, data formats are pluggable transformers that can transform messages from one form to another, and vice versa. Each data format is represented in Camel as an interface in `org.apache.camel.spi.DataFormat` containing two methods:

- `marshal`—For marshaling a message into another form, such as marshaling Java objects to XML, CSV, JSON, HL7, or other well-known data models
- `unmarshal`—For performing the reverse operation, which turns data from well-known formats back into a message

You may already have realized that these two functions are opposites; one is capable of reversing what the other has done, as illustrated in figure 3.6.

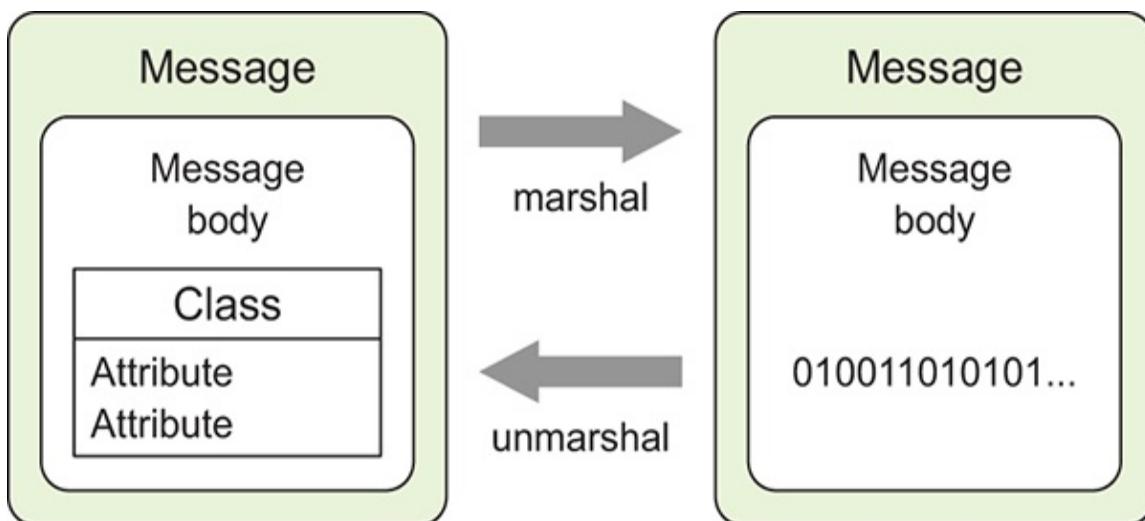


Figure 3.6 An object is marshaled to a binary representation; `unmarshal` can be used to get the object back.

We touched on data formats in section 3.3, where we covered XML transformations. This section covers data formats in more depth and using data types other than XML, such as CSV and JSON. We'll even look at how to create your own data formats.

We'll start our journey by briefly looking at the data formats Camel provides out of the box.

3.4.1 DATA FORMATS PROVIDED WITH CAMEL

Camel provides data formats for a range of well-known data models, some of which are listed in table 3.3.

Table 3.3 Selection of data formats provided out of the box with Camel

Data format	Data model	Artifact	Description
Avro	Binary Avro format	camel-avro	Supports serializing and deserializing messages by using Apache Avro
Base64	Base64 string	camel-base64	Can encode and decode into a base64 string
Bindy	CSV, FIX, fixed length	camel-bindy	Binds various data models to model objects by using annotations
Crypto	Any	camel-crypto	Encrypts and decrypts data by using the Java Cryptography Extension
CSV	CSV	camel-csv	Transforms to and from CSV by using the Apache Commons CSV library
GSON	JSON	camel-gson	Transforms to and from JSON by using the Google GSON library
GZip	Any	camel-gzip	Compresses and decompresses files (compatible with the popular gzip/gunzip tools)
HL7	HL7	camel-hl7	Transforms to and from HL7, which is a well-known data format in the health-care industry
JAXB	XML	camel-jaxb	Uses the JAXB 2.x standard for XML binding to and from Java objects

Jackson	JSON	camel-jackson	Transforms to and from JSON by using the ultra-fast Jackson library
PGP	Any	camel-crypto	Encrypts and decrypts data by using PGP
Protobuf	XML	camel-protobuf	Transforms to and from XML by using the Google Protocol Buffers library
SOAP	XML	camel-soap	Transforms to and from SOAP
Serialization	Object	camel-core	Uses Java Object Serialization to transform objects to and from a serialized stream
Syslog	RFC3164, RFC5424	camel-syslog	Transforms between RFC3164/RFC5424 messages and SyslogMessage model objects
XMLSecurity	XML	camel-xmlsecurity	Facilitates encryption and decryption of XML documents
XStream	XML	camel-xstream	Uses XStream for XML binding to and from Java objects
XStream	JSON	camel-xstream	Transforms to and from JSON by using the XStream library
Zip	Any	camel-core	Compresses and decompresses messages, and is most effective when dealing with large XML- or text-based payloads
Zip file	Zip file	camel-zipfile	Compresses and decompresses zip files

Camel provides more than 40 data formats out of the box. You can read more about these data formats at the Camel website (<http://camel.apache.org/data-format.html>). We've picked three to cover in the following section. They're among the most commonly used, and what you learn about those will also apply to the remainder of the data formats.

3.4.2 USING CAMEL'S CSV DATA FORMAT

The camel-csv data format is capable of transforming to and from CSV format. It uses Apache Commons CSV to do the work.

Suppose you need to consume CSV files, split out each row, and send it to a JMS queue. Sounds hard to do, but it's possible with little effort in a Camel route:

```
from("file:///rider/csvfiles")
    .unmarshal().csv()
    .split(body()).to("jms:queue:csv.record");
```

All you have to do is `unmarshal` the CSV files, which will read the file line by line and store all lines in the message body as a `java.util.List<List>` type. Then you use the splitter to split up the body, which will break the `java.util.List<List<String>>` into rows (each row represented as another `List<String>` containing the fields) and send each row to the JMS queue. You may not want to send each row as a `List` type to the JMS queue, so you can transform the row before sending, perhaps using a processor.

The same example in XML is a bit different, as shown here:

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file:///rider/csvfiles"/>
        <unmarshal><csv/></unmarshal>
        <split>
            <simple>body</simple>
            <to uri="jms:queue:csv.record"/>
        </split>
    </route>
```

```
</camelContext>
```

The noticeable difference is in the way you tell `<split>` that it should split up the message body. To do this, you need to provide `<split>` with an `Expression`, which is what the splitter should iterate when it performs the splitting. To do so, you can use Camel's built-in expression language called Simple (see appendix A), which knows how to do that.

NOTE The Splitter EIP is fully covered in chapter 5.

This example is in the source code for the book, in the `chapter3/order` directory. You can try the examples by running the following Maven goals:

```
mvn test -Dtest=PurchaseOrderCsvTest  
mvn test -Dtest=PurchaseOrderCsvSpringTest
```

At first, the data types that the CSV data format uses may seem confusing. They're listed in table 3.4.

Table 3.4 Data types that camel-csv uses when transforming to and from CSV format

Operation	From type	To type	Description
marshal	<code>Map<String, Object></code>	<code>OutputStream</code>	Contains a single row in CSV format.
marshal	<code>List<Map<String, Object>></code>	<code>OutputStream</code>	Contains multiple rows in CSV format; each row is separated by <code>\n</code> (newline).
unmarshal	<code>InputStream</code>	<code>List<List<String>></code>	Contains a <code>List</code> of rows; each row is another <code>List</code> of fields.

One problem with camel-csv is that it uses generic data types, such as `Map` or `List`, to represent CSV records. Often you'll already have model objects to represent your data in memory. Let's look at using model objects with the camel-bindy

component.

3.4.3 USING CAMEL'S BINDY DATA FORMAT

Two of the existing CSV-related data formats (camel-csv and camel-flatpack) are older libraries that don't take advantage of the new features in Java 1.5, such as annotations and generics. In light of this deficiency, Charles Moulliard stepped up and wrote the camel-bindy component to take advantage of these new possibilities. It's capable of binding CSV, FIX, and fixed-length formats to existing model objects by using annotations. This is similar to what JAXB does for XML.

Suppose you have a model object that represents a purchase order. By annotating the model object with camel-bindy annotations, you can easily transform messages between CSV and Java model objects, as shown in the following listing.

Listing 3.7 Model object annotated for CSV transformation

```
package camelinaction.bindy;

import java.math.BigDecimal;
import
org.apache.camel.dataformat.bindy.annotation.CsvRecord;
import
org.apache.camel.dataformat.bindy.annotation.DataField;

@CsvRecord(separator = ",", crlf = "UNIX") ❶
```

❶

Maps to CSV record

```
public class PurchaseOrder {
    @DataField(pos = 1) ❷
```

❷

Maps to column in CSV record

```
    private String name;
    @DataField(pos = 2, precision = 2) ❸
```

```

private BigDecimal price;
@DataField(pos = 3)    ②
private int amount;
}

```

First you mark the class with the `@CsvRecord` annotation ① to indicate that it represents a record in CSV format. Then you annotate the fields with `@DataField` according to the layout of the CSV record ②. Using the `pos` attribute, you can dictate the order in which they're output in CSV; `pos` starts with a value of 1. For numeric fields, you can additionally declare precision, which in this example is set to 2, indicating that the price should use two digits for cents. Bindy also has attributes for fine-grained layout of the fields, such as `pattern`, `trim`, and `length`. You can use `pattern` to indicate a data pattern, `trim` to trim the input, and `length` to restrict a text description to a certain number of characters.

Before you look at how to use Bindy in Camel routes, the data types Bindy expects to use are listed in table 3.5.

Table 3.5 Data types that Bindy uses when transforming to and from CSV format

Operation	From type	To type	Output description
marshal	<code>List<Map<String, Object>></code>	<code>OutputStream</code>	Contains multiple rows in CSV format; each row is separated by <code>\n</code> (newline).
unmarshal	<code>InputStream</code>	<code>List<Map<String, Object>></code>	Contains a <code>List</code> of rows; each row contains 1 ... n data models contained in a <code>Map</code> .

The important thing to notice in table 3.5 is that Bindy uses `Map<String, Object>` to represent a CSV row. At first, this may seem odd. Why doesn't it use a single model object for that? The answer is that you can have multiple model objects with the CSV record being scattered across those objects. For example, you could have fields 1 to 3 in one model object, fields 4 to 9 in another, and fields 10 to 12 in a third.

The map entry `<String, Object>` is distilled as follows:

- `Mapkey(String)`—Must contain the fully qualified class name of the model object
- `Mapvalue(Object)`—Must contain the model object

If this seems confusing, don't worry. The following listing should make it clearer.

Listing 3.8 Using Bindy to transform a model object to CSV format

```
public void testBindy() throws Exception {
    CamelContext context = new DefaultCamelContext();
    context.addRoutes(createRoute());
    context.start();
    MockEndpoint mock = context.getEndpoint("mock:result",
    MockEndpoint.class);
    mock.expectedBodiesReceived("Camel in
Action,69.99,1\n");
    PurchaseOrder order = new PurchaseOrder(); 1
```

1

Creates model object as usual

```
order.setAmount(1);
order.setPrice(new BigDecimal("69.99"));
order.setName("Camel in Action");
ProducerTemplate template =
context.createProducerTemplate();
template.sendBody("direct:toCsv", order); 2
```

2

Starts test

```
mock.assertIsSatisfied();
}

public RouteBuilder createRoute() {
    return new RouteBuilder() {
        public void configure() throws Exception {
```

```
from("direct:toCsv")
    .marshal().bindy(BindyType.Csv, ③
```

③

Transforms model object to CSV

```
camelInaction.bindy(PurchaseOrder.class) ③
        .to("mock:result");
    }
}
```

In [listing 3.8](#), you first create and populate the order model by using regular Java setters ①. Then you send the order model to the route by sending it to the `direct:toCsv` endpoint ② used in the route. The route will then marshal the order model to CSV by using Bindy ③. Notice that Bindy is configured to use CSV mode via `BindType.Csv`. To let Bindy know how to map the order model object, you need to provide a class annotated with Bindy annotations, as in [listing 3.7](#).

NOTE Listing 3.8 uses `MockEndpoint` to easily test that the CSV record is as expected. Chapter 9 covers testing with Camel, and you'll learn all about using `MockEndpoint`.

You can try this example from the `chapter3/order` directory by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderBindyTest
```

The source code for the book also contains a *reverse* example of how to use Bindy to transform a CSV record into a Java object. You can try it by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderUnmarshalBindyTest
```

CSV is only one of the well-known data formats that Bindy supports. Bindy is equally capable of working with fixed-length

and FIX data formats, both of which follow the same principles as CSV.

It's now time to leave CSV and look at a more modern format: JSON.

3.4.4 USING CAMEL'S JSON DATA FORMAT

JavaScript Object Notation (JSON) is a data-interchange format, and Camel provides six components that support the JSON data format: camel-xstream, camel-gson, camel-jackson, camel-boon, camel-fastjson, camel-johnzon. This section focuses on camel-jackson because Jackson is a popular JSON library.

Back at Rider Auto Parts, you now have to implement a new service that returns order summaries rendered in JSON format. Doing this with Camel is fairly easy, because Camel has all the ingredients needed to brew this service. The following listing shows how to ramp up a prototype.

Listing 3.9 An HTTP service that returns order summaries rendered in JSON format

```
<bean id="orderService"
      class="camelinaction.OrderServiceBean"/>

<camelContext id="camel"
      xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
      <json id="json" library="Jackson"/> 1
    </dataFormats>
    <route>
```

1

Sets up JSON data format

```
      <from uri="jetty://http://0.0.0.0:8080/order"/>
      <bean ref="orderService" method="lookup"/> 2
```

2

Invokes bean to retrieve data for reply

```
<marshal ref="json"/>
</route>
</camelContext>
```

First you need to set up the JSON data format and specify that the Jackson library should be used ❶. Then you define a route that exposes the HTTP service using the Jetty endpoint. This example exposes the Jetty endpoint directly in the URI. By using `http://0.0.0.0:8080/order`, you tell Jetty that any client can reach this service on port 8080. Whenever a request hits this HTTP service, it's routed to the `orderService` bean ❷, and the `lookup` method is invoked on that bean. The result of this bean invocation is then marshaled to JSON format and returned to the HTTP client.

The order service bean could have a method signature such as this:

```
public PurchaseOrder lookup(@Header(name = "id") String id)
```

This signature allows you to implement the lookup logic as you wish. You'll learn more about the `@Header` annotation in chapter 4, when we cover how bean parameter binding works in Camel.

Notice that the service bean can return a POJO that the JSON library is capable of marshaling. For example, suppose you used the `PurchaseOrder` from [listing 3.7](#) and had JSON output as follows:

```
{"name": "Camel in Action", "amount": 1.0, "price": 69.99}
```

The HTTP service itself can be invoked by an `HTTP Get` request with the `id` of the order as a parameter:

```
http://0.0.0.0:8080/order/service?id=123.
```

Notice how easy it is with Camel to bind the HTTP `id` parameter as the `String id` parameter with the help of the `@Header` annotation.

You can try this example yourself from the `chapter3/order` directory by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderJSONTest
```

So far, we've used data formats with their default settings. But what if you need to configure the data format, for example, to use another splitter character with the CSV data format? That's the topic of the next section.

3.4.5 CONFIGURING CAMEL DATA FORMATS

In section 3.4.2, you used the CSV data format, but this data format offers many additional settings. The following listing shows how to configure the CSV data format.

Listing 3.10 Configuring the CSV data format

```
public void configure() {  
    CsvDataFormat myCsv = new CsvDataFormat()  
        .setDelimiter(',')      ①
```

①

Creates and configures a custom CSV data format

```
        .setHeader(new String[]{      ①  
            "id", "customerId", "date", "item", "amount",  
            "description"});      ①  
  
        from("direct:toCsv")  
            .marshal(myCsv)      ②
```

②

Uses CSV data format

```
        .to("file://acme/outbox/csv");  
}
```

Configuring data formats in Camel is typically done directly on `DataFormat` itself; sometimes you may also need to use the API that the third-party library under the hood provides. In [listing 3.10](#), the CSV data format nicely wraps the third-party API so you can just configure the `DataFormat` directly ①. Here you set

the semicolon as a delimiter and specify the order of the fields ❶. The use of the data format stays the same, so all you need to do is refer to it from the `marshal` ❷ or `unmarshal` methods. This same principle applies to all data formats in Camel.

TIP You can learn how to create your own data format in chapter 8, section 8.5.

You've learned all about data formats, and now it's time to say goodbye to data formats and take a look at using templating with Camel for data transformation. Templating is extremely useful when you need to generate automatic reply emails.

3.5 *Transforming with templates*

Camel provides slick integration with two template languages:

- *Apache Velocity*—Probably the best-known templating language (<http://camel.apache.org/velocity.html>)
- *Apache FreeMarker*—Another great templating language from Apache (<http://camel.apache.org/freemarker.html>)

These two templating languages are fairly similar to use, so we discuss only Velocity here.

3.5.1 **USING APACHE VELOCITY**

Rider Auto Parts has implemented a new order system that must send an email reply when a customer has submitted an order. Your job is to implement this feature.

The reply email could look like this:

Dear customer
Thank you for ordering X piece(s) of XXX at a cost of XXX.
This is an automated email, please do not reply.

Three pieces of information in the email must be replaced at

runtime with real values. You need to adjust the email to use the Velocity template language, and then place it into the source repository as `src/test/resources/email.vm`:

```
Dear customer
Thank you for ordering ${body.amount} piece(s) of
${body.name} at a cost of ${body.price}.
This is an automated email, please do not reply.
```

Notice that you insert `{}` placeholders in the template, which instructs Velocity to evaluate and replace them at runtime. Camel prepopulates the Velocity context with numerous entities that are then available to Velocity. Those entities are listed in table 3.6.

NOTE The entities in table 3.6 also apply to other templating languages, such as FreeMarker.

Table 3.6 Entities that are prepopulated in the Velocity context and available at runtime

Entity	Type	Description
camelContext	org.apache.camel.CamelContext	The CamelContext.
exchange	org.apache.camel.Exchange	The current exchange.
in	org.apache.camel.Message	The input message. This can clash with a reserved word in some languages; use request instead.
request	org.apache.camel.Message	The input message.
body	java.lang.Object	The input message body.
headers	java.util.Map	The input message headers.
response	org.apache.camel.Message	The output message.

out	org.apache.camel.Message	The output message. This can clash with a reserved word in some languages; use response instead.
-----	--------------------------	--

Using Velocity in a Camel route is as simple as this:

```
from("direct:sendMail")
    .setHeader("Subject", constant("Thanks for ordering"))
    .setHeader("From", constant("donotreply@riders.com"))
    .to("velocity://rider/mail.vm")
    .to("smtp://mail.riders.com?
user=camel&password=secret");
```

All you have to do is route the message to the Velocity endpoint that's configured with the template you want to use, which is the rider/mail.vm file that's loaded from the classpath by default. All the template components in Camel use the same resource loader, which allows you to load templates from the classpath, file paths, and other such locations. You can use the same prefixes listed in [table 3.2](#).

You can try this example by going to the chapter3/order directory in the book's source code and running the following Maven goal:

```
mvn test -Dtest=PurchaseOrderVelocityTest
```

We'll now leave data transformation and look at type conversion. Camel has a powerful type-converter mechanism that removes all need for boilerplate type-converter code.

3.6 Understanding Camel type converters

Camel provides a built-in type-converter system that automatically converts between well-known types. This system allows Camel components to easily work together without having type mismatches. And from the Camel user's perspective, type conversions are built into the API in many places without being invasive. For example, you used it in [listing 3.1](#):

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

The `getBody` method is passed the type you want to have returned. Under the covers, the type-converter system converts the returned type to a `String` if needed.

In this section, you'll take a look at the insides of the type-converter system. We'll explain how Camel scans the classpath on startup to register type converters dynamically. We'll also show how to use it from a Camel route, and how to build your own type converters.

3.6.1 HOW THE CAMEL TYPE-CONVERTER MECHANISM WORKS

To understand the type-converter system, you first need to know what a type converter in Camel is. [Figure 3.7](#) illustrates the relationship between `TypeConverterRegistry` and the `TypeConverters` it holds.

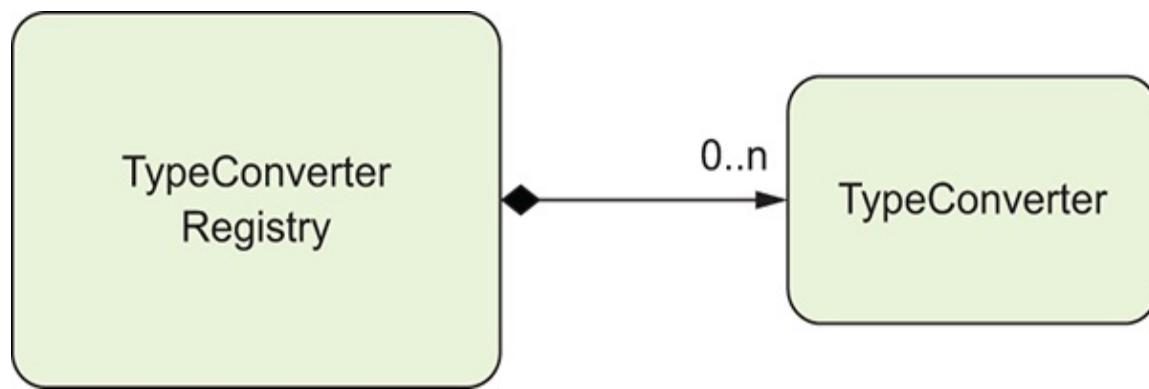


Figure 3.7 The `TypeConverterRegistry` contains many `TypeConverters`

`TypeConverterRegistry` is where all the type converters are registered when Camel is started. At runtime, Camel uses the `TypeConverterRegistry`'s `lookup` method to look up a suitable `TypeConverter`:

```
TypeConverter lookup(Class<?> toType, Class<?> fromType);
```

By using `TypeConverter`, Camel can then convert one type to another by using `TypeConverter`'s `convertTo` method, which is defined as follows:

```
<T> T convertTo(Class<T> type, Object value);
```

NOTE Camel implements about 350 or more type converters out of the box, which are capable of converting to and from the most commonly used types.

LOADING TYPE CONVERTERS INTO THE REGISTRY

On startup, Camel loads all the type converters into the `TypeConverterRegistry` by using a classpath-scanning solution. This allows Camel to pick up type converters not only from `camel-core`, but also from any of the other Camel components, including your Camel applications. You'll see this in section 3.6.3 when you build your own type converter.

Camel uses

`org.apache.camel.impl.converter.AnnotationTypeConverterLoader` to scan and load the type converters. To avoid scanning zillions of classes, it reads a service discovery file in the `META-INF` folder: `META-INF/services/org/apache/camel/TypeConverter`. This is a plain-text file that has a list of fully qualified class names and packages that contain Camel type converters. The special file is needed to avoid scanning every possible JAR and all their packages, which would be time-consuming. This special file tells Camel whether the JAR file contains type converters. For example, the file in `camel-cxf` contains the following entries:

```
org.apache.camel.component.cxf.converter.CxfConverter  
org.apache.camel.component.cxf.converter.CxfPayloadConverte
```

`AnnotationTypeConverterLoader` loads those classes that have been annotated with `@Converter`, and then searches within them for public methods that are annotated with `@Converter`. Each of those methods is considered a type converter. Yes, the class `@Converter` annotation is a bit of overkill when we've already defined the class name in the `TypeConverterText` file. We need this because we can also specify package names, which could include many classes. For example, a package name of

`org.apache.camel.component.cxf.converter` also could have been provided in the `TypeConverter` text file and would have included `CxfConverter` and `CxfPayloadConverter`. Using the fully qualified class name is preferred, though, because Camel loads them more quickly.

This process is best illustrated with an example. The following code is a snippet from the `IOConverter` class from the camel-core JAR:

```
@Converter  
public final class IOConverter {  
    @Converter  
    public static InputStream toInputStream(URL url) throws  
    IOException {  
        return IOHelper.buffered(url.openStream());  
    }  
}
```

Camel will go over each method annotated with `@Converter` and look at the method signature. The first parameter is the *from* type, and the return type is the *to* type. In this example, you have a `TypeConverter` that can convert from a `URL` to an `InputStream`. By doing this, Camel loads all the built-in type converters, including those from the Camel components in use.

TIP Type converters can also be loaded into the registry manually. This is often useful if you need to quickly add a type converter into your application or want full control over when it will be loaded. You can find more information in the online documentation (<http://camel.apache.org/type-converter.html>).

Now that you know how the Camel type converters are loaded, let's look at using them.

3.6.2 USING CAMEL TYPE CONVERTERS

As we mentioned, the Camel type converters are used throughout Camel, often automatically. But you might want to

use them to force a specific type to be used in a route, such as before sending data back to a caller or a JMS destination. Let's look at how to do that.

Suppose you need to route files to a JMS queue by using `javax.jms.TextMessage`. To do so, you can convert each file to a `String`, which forces the JMS component to use `TextMessage`. This is easy to do in Camel—you use the `convertBodyTo` method, as shown here:

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

If you're using the XML DSL, you provide the type as an attribute instead, like this:

```
<route>
    <from uri="file://riders/inbox"/>
    <convertBodyTo type="java.lang.String"/>
    <to uri="jms:queue:inbox"/>
</route>
```

You can omit the `java.lang.` prefix on the type, which can shorten the syntax: `<convertBodyTo type="String"/>`.

Another reason for using `convertBodyTo` is to read files by using a fixed encoding such as `UTF-8`. This is done by passing in the encoding as the second parameter:

```
from("file://riders/inbox")
    .convertBodyTo(String.class, "UTF-8")
    .to("jms:queue:inbox");
```

TIP If you have trouble with a route because of the payload or its type, try using `.convertBodyTo(String.class)` at the start of the route to convert to a `String` type, which is a well-supported type. If the payload can't be converted to the desired type, a `NoTypeConversionAvailableException` exception is thrown.

That's all there is to using type converters in Camel routes.

Before we wrap up this chapter, though, let's take a look at how to write your own type converter.

3.6.3 WRITING YOUR OWN TYPE CONVERTER

Writing your own type converter is easy in Camel. You already saw what a type converter looks like in section 3.6.1, when you looked at how type converters work.

Suppose you want to write a custom type converter that can convert a `byte[]` into a `PurchaseOrder` model object (an object you used in [listing 3.7](#)). As you saw earlier, you need to create an `@Converter` class containing the type-converter method, as shown in the following listing.

Listing 3.11 A custom type converter to convert from `byte[]` to `PurchaseOrder` type

```
@Converter
public final class PurchaseOrderConverter
    @Converter
        public static PurchaseOrder toPurchaseOrder(byte[]
data,
                                                Exchange
exchange) {
    TypeConverter converter = exchange.getContext()
    .getTypeConverter(); 1
```

1

Grabs `TypeConverter` to reuse

```
String s = converter.convertTo(String.class, data);
if (s == null || s.length() < 30) {
    throw new IllegalArgumentException("data is
invalid");
}
s = s.replaceAll("##START##", ""); 2
```

2

Converts from `String` to `PurchaseOrder`

```

    s = s.replaceAll("##END##", "");      ②
    String name = s.substring(0, 9).trim();  ②

    String s2 = s.substring(10, 19).trim();  ②
    BigDecimal price = new BigDecimal(s2);  ②

    price.setScale(2);                    ②
    String s3 = s.substring(20).trim();    ②
    Integer amount = converter          ②
        .convertTo(Integer.class,
s3);  ②
    return new PurchaseOrder(name, price,
amount);  ②
}
}

```

The Exchange gives you access to the camelContext and thus to the parent TypeConverter ①, which you use in this method to convert between strings and numbers. The rest of the code is the logic for parsing the custom protocol and returning the PurchaseOrder ②. Notice that you can use converter to easily convert between well-known types.

All you need to do now is add the service discovery file, named TypeConverter, in the META-INF directory. As explained previously, this file contains the fully qualified name of the @Converter class.

If you cat the TypeConverter file, you'll see this:

```
$ cat src/main/resources/META-
INF/services/org/apache/camel/TypeConverter
camelinaction.PurchaseOrderConverter
```

This example can be found in the chapter3/converter directory of the book's source code, which you can try by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderConverterTest
```

RETURNING NULL VALUES

By default, a null return value from a type converter isn't valid. Camel considers null as a "miss" and adds the pair of types

you're trying to convert to a blacklist so they won't be tried again. For example, if our previous example returned `null`, the conversion from `byte[]` to `PurchaseOrder` would be blacklisted. If `null` is a valid return value for your conversion, you can force Camel to accept it by using the `allowNull` option on the `@Converter` annotation. For example, if the example in [listing 3.11](#) required a `null` return value, you could do something like this:

```
@Converter(allowNull = true)
public static PurchaseOrder toPurchaseOrder(byte[] data,
                                             Exchange
exchange) {
    ...
}
```

ADDING TYPE CONVERTERS TO CAMEL-CORE

If you're on track to becoming a star Camel rider and want to write a shiny new type converter for the camel-core module, you may notice that type-converter loading is handled differently there. The META-INF/services/org/apache/camel/TypeConverter file specifies the `org.apache.camel.core` package, which doesn't exist. It's just a dummy package name. The type converters for camel-core are specified directly in `org.apache.camel.impl.converter.CorePackageScanClassResolver`, and you can add them there. And that completes this chapter on transforming data with Camel.

3.7 Summary and best practices

Data transformation is the cornerstone of any integration kit; it bridges the gap between various data types and formats. It's also essential in today's industry, because more and more disparate systems need to be integrated to support the ever-changing businesses and world we live in.

This chapter covered many of the possibilities Camel offers for data transformation. You learned how to format messages by using EIPs and beans. You also learned that Camel provides

special support for transforming XML documents by using XSLT components and XML-capable data formats. Camel provides data formats for well-known data models, which you learned to use, and it even allows you to build your own data formats. We also took a look into the templating world, which can be used to format data in specialized cases, such as generating email bodies. Finally, we looked at how the Camel type-converter mechanism works and learned that it's used internally to help all the Camel components work together. You learned how to use it in routes and how to write your own converters.

Here are a few key tips you should take away from this chapter:

- *Data transformation is often required*—Integrating IT systems often requires you to use different data formats when exchanging data. Camel can act as the mediator and has strong support for transforming data in any way possible. Use the various features in Camel to aid with your transformation needs.
- *Java is powerful*—Using Java code isn't a worse solution than using a fancy mapping tool. Don't underestimate the power of the Java language. Even if it takes 50 lines of grunt boilerplate code to get the job done, you have a solution that can easily be maintained by fellow engineers.
- *Prefer to use beans over processors*—If you're using Java code for data transformation, you can use beans or processors. Processors are more dependent on the Camel API, whereas beans allow loose coupling. Chapter 4 covers how to use beans.

This chapter, along with chapter 2, covered two crucial features of integration kits: routing and transformation. The next chapter dives into the world of Java beans, and you'll see how Camel can easily adapt to and use your existing beans. This allows a higher degree of reuse and loose coupling, so you can keep your business and integration logic clean and apart from Camel and other middleware APIs.

4

Using beans with Camel

This chapter covers

- Calling Java beans with Camel
- Understanding the Service Activator EIP
- How Camel looks up beans using registries
- How Camel selects bean methods to invoke
- Using bean parameter bindings
- Using Java beans as predicates or expressions in routes

If you've been developing software for many years, you've likely worked with various component models, such as CORBA, EJB, JBI, SCA, and lately OSGi. Some of these, especially the earlier ones, imposed a great deal on the programming model, dictating what you could and couldn't do, and they often required complex packaging and deployment procedures. This left the everyday programmer with a lot of concepts to learn and master. In some cases, much more time was spent working around the restrictive programming and deployment models than on the business application itself.

Because of this growing complexity and the resulting frustrations, a simpler, more pragmatic programming model arose from the open source community: the Plain Old Java Object (POJO) model. Many open source projects have proven

that the POJO programming model and a lightweight container meet the expectations of today's businesses. In fact, the simple programming model and lightweight container concept proved superior to the heavyweight and overly complex enterprise application and integration servers that were used before. This trend continues to this day, and we're seeing the rise of microservices and containerless deployments. We cover microservices in chapter 7, but first you need to learn more about Camel basics, including this chapter's topic of using Java beans with Camel.

So what about Camel? Well, Camel doesn't mandate using a specific component or programming model. It doesn't mandate a heavy specification that you must learn and understand to be productive. Camel doesn't require you to repackage any of your existing libraries or require you to use the Camel API to fulfill your integration needs. Camel is on the same page as the Spring Framework; both are lightweight containers favoring the POJO programming model. Camel recognizes the power of the POJO programming model and goes to great lengths to work with your beans. By using beans, you fulfill an important goal in the software industry: reducing coupling. Camel not only offers reduced coupling with beans, but you get the same loose coupling with Camel routes. For example, three teams can work simultaneously on their own sets of routes, which can easily be combined into one system.

This chapter starts by showing you how *not* to use beans with Camel, which will make it clearer how you *should* use beans. After that, you'll learn the theory behind the Service Activator EIP and dive inside Camel to see how this pattern is implemented. We then cover the bean-binding process, which gives you fine-grained control over binding information to the parameters on the invoked method from within Camel and the currently routed message. The chapter ends by explaining how to use beans as predicates and expressions in your route.

4.1 Using beans the hard way and the easy

way

In this section, you'll walk through an example that shows how *not* to use beans with Camel—the hard way to use beans. Then you'll look at how to use beans the easy way.

Suppose you have an existing bean that offers an operation (a service) you need to use in your integration application. For example, `HelloBean` offers the `hello` method as its service:

```
public class HelloBean {  
    public String hello(String name) {  
        return "Hello " + name;  
    }  
}
```

Let's look at various ways to use this bean in your application.

4.1.1 INVOKING A BEAN FROM PURE JAVA

By using a Camel Processor, you can invoke a bean from Java code, as in the following listing.

Listing 4.1 Using a Processor to invoke the hello method on the HelloBean

```
public class InvokeWithProcessorRoute extends RouteBuilder {  
  
    public void configure() throws Exception {  
        from("direct:hello")  
            .process(new Processor() {  
                ①
```

1

Uses a Processor

```
                public void process(Exchange exchange) throws  
Exception {  
                    String name =  
exchange.getIn().getBody(String.class);  
                ②
```

2

Extracts input from Camel message

```
HelloBean hello = new HelloBean();
String answer = hello.hello(name);
```

3

Invokes HelloBean

```
exchange.getOut().setBody(answer);
```

4

Response from HelloBean is set on OUT message

```
}
```

```
});
```

```
}
```

```
}
```

Listing 4.1 shows a `RouteBuilder`, which defines the route. You use an inlined Camel Processor, which gives you the `process` method, in which you can work on the message with Java code 1. First, you must extract the message body from the input message 2, which is the parameter you'll use when you invoke the bean later. Then you need to instantiate the bean and invoke it 3. Finally, you must set the output from the bean on the output message 4.

Now that you've done it the hard way using the Java DSL, let's take a look at using XML DSL.

4.1.2 INVOKING A BEAN DEFINED IN XML DSL

When using Spring XML or OSGi Blueprint as a bean container, the beans are defined using its XML files. Listings [4.2](#) and [4.3](#) show how to revise [listing 4.1](#) to work with a Spring bean this way.

Listing 4.2 Setting up Spring to use a Camel route that uses HelloBean

```
<bean id="helloBean"  
class="camelaction.HelloBean"/>
```

①

①

Defines HelloBean

```
<bean id="route"  
class="camelaction.InvokeWithProcessorSpringRoute"/>  
  
<camelContext id="camel"  
xmlns="http://camel.apache.org/schema/spring">  
    <routeBuilder ref="route"/>  
</camelContext>
```

First you define `HelloBean` in the Spring XML file with the ID `helloBean` 1. You still want to use the Java DSL to build the route, so you need to declare a bean that contains the route. Finally, you define `camelContext`, which is the way you get Spring and Camel to work together.

[Listing 4.3](#) takes a closer look at the route.

Listing 4.3 A Camel route using a Processor to invoke HelloBean

```
public class InvokeWithProcessorSpringRoute extends  
RouteBuilder {  
    @Autowired ①
```

①

Injects HelloBean

```
private HelloBean hello;  
  
public void configure() throws Exception {  
    from("direct:hello")  
        .process(new Processor() {  
            public void process(Exchange exchange) throws  
Exception {  
                String name =  
exchange.getIn().getBody(String.class);  
                String answer = hello.hello(name); ②
```

2

Invokes HelloBean

```
        exchange.getOut().setBody(answer);
    }
});
}
```

The route in [listing 4.3](#) is nearly identical to the route in [listing 4.1](#). The difference is that now the bean is dependency injected using the Spring @Autowired annotation 1, and instead of instantiating the bean, you use the injected bean directly 2.

You can try these examples on your own; they're in the chapter4/bean directory of the book's source code. Run Maven with these goals to try the last two examples:

```
mvn test -Dtest=InvokeWithProcessorTest  
mvn test -Dtest=InvokeWithProcessorSpringTest
```

So far, you've seen two examples of using beans with a Camel route, and a bit of plumbing is required to get it all to work. As you've seen, it's hard to work with beans for the following reasons:

- You must use Java code to invoke the bean.
- You must use the Camel Processor, which clutters the route, making it harder to understand what happens (route logic is mixed in with implementation logic).
- You must extract data from the Camel message and pass it to the bean, and you must move any response from the bean back into the Camel message.
- You must instantiate the bean yourself, or have it dependency injected.

Now let's look at the easy way of doing it.

4.1.3 USING BEANS THE EASY WAY

Calling a bean in Camel is easy. The previous example in [listing 4.3](#) can be reduced to a few lines of code, as shown in the following listing.

Listing 4.4 Camel route using bean to invoke HelloBean

```
public class InvokeWithBeanSpringRoute extends RouteBuilder
{
    @Autowired          ①

    ①
    Injects HelloBean

    private HelloBean helloBean;

    public void configure() throws Exception {
        from("direct:hello")
            .bean(helloBean, "hello");      ②

    ②
    Invokes HelloBean

}
}
```

Now the route is much shorter—only two lines of code. You still let Spring dependency inject `HelloBean` 1. And this time, the Camel route doesn't use a Processor to invoke the bean but instead uses Camel's bean method 2:

```
.bean(helloBean, "hello")
```

The first parameter is the bean to call, and `hello` is the name of the method to call, which means Camel will invoke the `hello` method on the bean instance named `helloBean`.

That's a staggering reduction from seven lines of Processor code to one line with bean. And on top of that, the one code line is much easier to understand. It's all high-level abstraction, containing no low-level code details, which were required when using inlined Processors.

In XML DSL, this is just as easy. The example can be written as follows:

```
<bean id="helloBean" class="camelaction.HelloBean"/>
```

Injects HelloBean

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
<route>
<from uri="direct:start"/>
<bean ref="helloBean" method="hello"/>
```

Invokes HelloBean

```
</route>
</camelContext>
```

Prefer to specify the method name

If the bean has only a single method, you can omit specifying the method name, and the example can be reduced to this:

```
from("direct:hello").bean("helloBean");
```

In XML DSL, the example looks like this:

```
<bean ref="helloBean"/>
```

Omitting the method name is recommended only when the beans have a single method. When a bean has several methods, Camel has to find the most suitable method to invoke based on a series of factors. You'll dive into how this works in section 4.4.2. As a rule of thumb, it's better to specify the method name, which also lets humans who maintain or later have to modify your code understand exactly which method is being invoked.

We've provided these examples with the source code located in the chapter4/beans directory. You can run the example by using the following Maven goals:

```
mvn test -Dtest=InvokeWithBeanTest  
mvn test -Dtest=InvokeWithBeanSpringTest
```

TIP In the Java DSL, you don't have to preregister the bean in the registry. Instead, you can provide the class name of the bean, and Camel will instantiate the bean on startup. The previous example could be written like this:

```
from("direct:hello").bean(HelloBean.class);.
```

Camel also allows you to call beans by using the bean component as an endpoint.

USING TO INSTEAD OF BEAN

When you start learning Camel and build your first set of Camel routes, you often start with a few EIP patterns and can get far with just using `from` and `to`. Camel also makes it easy to call Java beans by using the `to` style. The previous examples could have been written as follows:

```
from("direct:hello")  
    .to("bean:helloBean?method=hello");
```

And in XML DSL:

```
<route>  
    <from uri="direct:start"/>  
    <to uri="bean:helloBean?method=hello"/>  
</route>
```

You may ask, when should you use `bean` and when should you use `to`? We, the authors, have no preference. Both are equally good to use. Sometimes you're most comfortable building routes that are only using `from` and `to`. And other times you use a lot

more EIP patterns, and bean fits better. The only caveat with using to is that the bean must be specified by using a String value as either the fully qualified class name or the bean name. Using the fully qualified class name is more verbose (especially if you have long package names):

```
from("direct:hello")
    .to("bean:camelaction.HelloBean?method=hello");
```

And in XML DSL:

```
<route>
    <from uri="direct:start"/>
    <to uri="bean:camelaction.HelloBean?method=hello"/>
</route>
```

Now let's look at how to work with beans in Camel from the EIP perspective.

4.2 Understanding the Service Activator pattern

The Service Activator pattern is an enterprise pattern described in Hohpe and Woolf's *Enterprise Integration Patterns* book. It describes a service that can be invoked easily from both messaging and non-messaging services. [Figure 4.1](#) illustrates this principle.

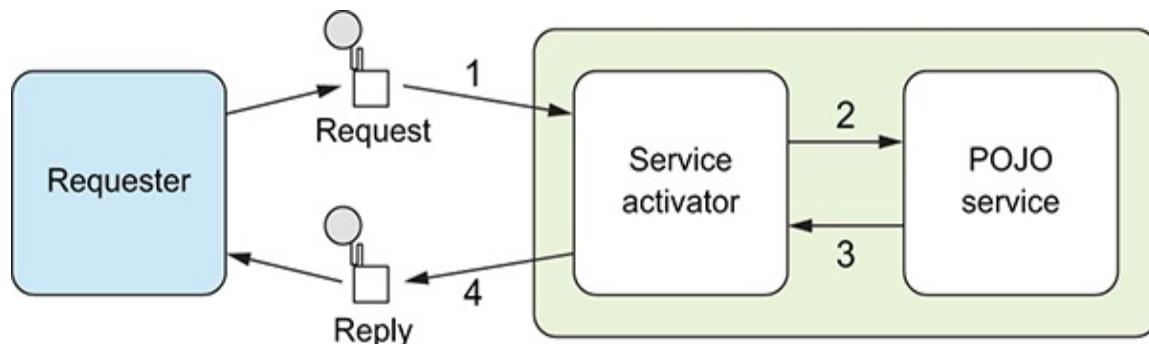


Figure 4.1 The service activator mediates between the requestor and the POJO service.

This service activator component invokes a service based on an

incoming request and returns an outbound reply. The service activator acts as a mediator between the requester and the POJO service. The requester sends a request to the service activator 1, which is responsible for adapting the request to a format the POJO service understands (mediating) and passing the request on to the service 2. The POJO service then returns a reply to the service activator 3, which passes it back (requiring no translation on the way back) to the waiting requester 4.

As you can see in [figure 4.1](#), the service is the POJO, and the service activator is something in Camel that can adapt the request and invoke the service. That something is the Camel Bean component, which eventually uses `org.apache.camel.component.bean.BeanProcessor` to do the work. You'll look at how this `BeanProcessor` works in section 4.4. You should regard the Camel Bean component as the Camel implementation of the Service Activator pattern.

Compare the Service Activator pattern in [figure 4.1](#) to the Camel route example you looked at in section 4.1.3, as illustrated in [figure 4.2](#).

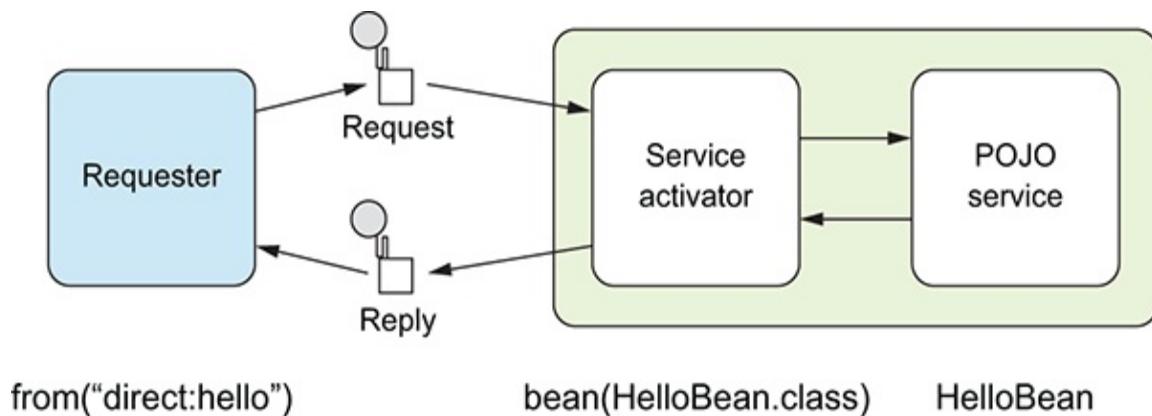


Figure 4.2 Relationship between a Camel route and the Service Activator EIP

Here you can see how the Camel route maps to the Service Activator EIP. The requester is the node that comes before the bean—it's the `from("direct:hello")` in our example. The service activator itself is the bean node, which is represented by the bean component in Camel. And the POJO service is the `HelloBean` bean itself.

You now know the theory behind how Camel works with beans—the Service Activator pattern. But before you can use a bean, you need to know where to look for it. This is where the registry comes into the picture. Let's look at how Camel works with various registries.

4.3 Using Camel's bean registries

When Camel works with beans, it looks them up in a registry to locate them. Camel's philosophy is to use the best of the available frameworks, so it uses a pluggable registry architecture to integrate them. Spring is one such framework, and [figure 4.3](#) illustrates how the registry works.

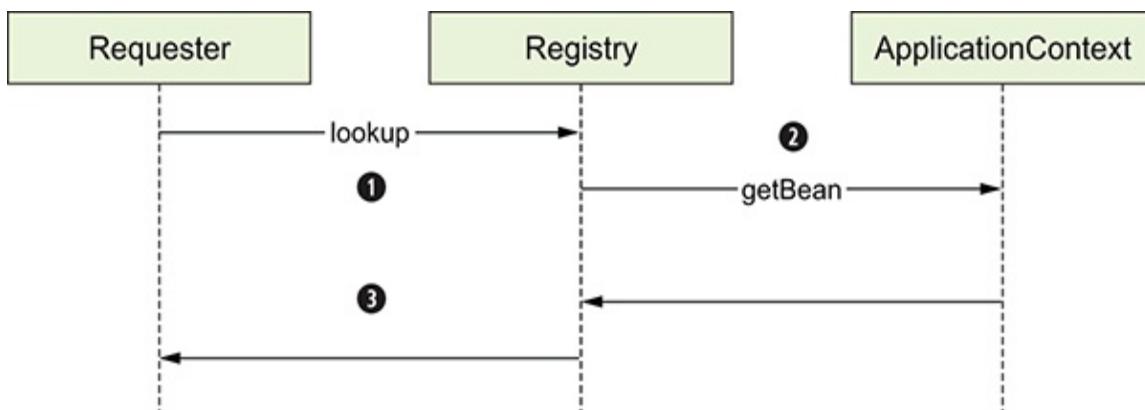


Figure 4.3 A requester looks up a bean using the Camel registry, which then uses the Spring ApplicationContext to determine where the bean resides.

The Camel registry is an abstraction that sits between the caller and the real registry. When a caller (requester) needs to look up a bean 1, it uses the Camel Registry. The Camel Registry then does the lookup via the real registry 2. The bean is then returned to the requester 3 via the Camel Registry. This structure allows loose coupling but also a pluggable architecture that integrates with multiple registries. All the requester needs to know is how to interact with the Camel Registry.

The registry in Camel is merely a Service Provider Interface (SPI) defined in the `org.apache.camel.spi.Registry` interface, as follows:

```
Object lookupByName(String name);  
<T> T lookupByNameAndType(String name, Class<T> type);  
<T> Map<String, T> findByTypeWithNamed(Class<T> type);  
<T> Set<T> findByType(Class<T> type);
```

You'll most often use one of the first two methods to look up a bean by its name. For example, to look up `HelloBean`, you'd do this:

```
HelloBean hello = (HelloBean) context.getRegistry()  
    .lookupByName("helloBean");
```

To get rid of that ugly typecast, you can use the second method instead:

```
HelloBean hello = context.getRegistry()  
    .lookupByNameAndType("helloBean",  
HelloBean.class);
```

NOTE The second method offers type-safe lookups because you provide the expected class as the second parameter. Under the hood, Camel uses its type-converter mechanism to convert the bean to the desired type, if necessary.

The last two methods, `findByTypeWithNamed` and `findByType`, are mostly used internally by Camel to support convention over configuration—they allow Camel to look up beans by type without knowing the bean name.

The registry itself is an abstraction and thus an interface. Table 4.1 lists the four implementations shipped with Camel.

Table 4.1 Registry implementations shipped in Camel

Regist ry	Description
JndiRegis try	An implementation that uses an existing Java Naming and Directory Interface (JNDI) registry to look up beans.
Simple Regis try	An in-memory-only registry that uses <code>java.util.Map</code> to hold the entries.

ApplicationContext Registry	An implementation that works with Spring to look up beans in the Spring ApplicationContext. This implementation is automatically used when you're using Camel in a Spring environment.
OsgiServiceRegistry	An implementation capable of looking up beans in the OSGi service reference registry. This implementation is automatically used when using Camel in an OSGi environment.
BlueprintContainerRegistry	An implementation that works with OSGi Blueprint to look up beans from the OSGi service registry as well as in the Blueprint container. This implementation is automatically used when you're using Camel in an OSGi Blueprint environment.
CdiBeanRegistry	An implementation that works with CDI to look up beans in the CDI container. This implementation is used when using the camel-cdi component.

The following sections go over each of these six registries.

4.3.1 JNDIREGISTRY

JndiRegistry, as its name indicates, integrates with a JNDI-based registry. It was the first registry that Camel integrated, so it's also the default registry if you create a Camel instance without supplying a specific registry, as this code shows:

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

JndiRegistry (like SimpleRegistry) is often used for testing or when running Camel standalone. Many of the unit tests in Camel use JndiRegistry because they were created before SimpleRegistry was added to Camel.

NOTE JndiRegistry will be replaced with SimpleRegistry as the default registry in Camel 3.0 onward.

The source code for this book contains an example of using JndiRegistry that's identical to the next example using

SimpleRegistry (shown in [listing 4.5](#)). We recommend that you read the next section and then compare the two examples.

You can try this test by going to the chapter4/bean directory and running this Maven goal:

```
mvn test -Dtest=JndiRegistryTest
```

Now let's look at the next registry: SimpleRegistry.

4.3.2 SIMPLEREGISTRY

SimpleRegistry is a Map-based registry that's used for testing or when running Camel standalone. For example, if you wanted to unit-test the HelloBean example, you could use SimpleRegistry to enlist HelloBean and refer to it from the route.

Listing 4.5 Using SimpleRegistry to unit-test a Camel route

```
public class SimpleRegistryTest extends TestCase {  
  
    private CamelContext context;  
    private ProducerTemplate template;  
  
    protected void setUp() throws Exception {  
        SimpleRegistry registry = new SimpleRegistry();  
        registry.put("helloBean", new HelloBean()); ①  
    }  
}
```

1

Registers HelloBean in SimpleRegistry

```
context = new DefaultCamelContext(registry); ②
```

2

Uses SimpleRegistry with Camel

```
template = context.createProducerTemplate();  
context.addRoutes(new RouteBuilder() {  
    public void configure() throws Exception {  
        from("direct:hello").bean("helloBean", "hello");  
    }  
});
```

```
    }
});
context.start();
}

protected void tearDown() throws Exception {
    template.stop(); ③
```

③

Cleans up resources after test

```
    context.stop(); ③
}

public void testHello() throws Exception {
    Object reply = template.requestBody("direct:hello",
"World");
    assertEquals("Hello World", reply);
}
```

First you create an instance of `SimpleRegistry` and populate it with `HelloBean` under the `helloBean` name 1. Then, to use this registry with Camel, you pass the registry as a parameter to the `DefaultCamelContext` constructor 2. To aid when testing, you create `ProducerTemplate`, which makes it simple to send messages to Camel, as you can see in the test method. Finally, when the test is done, you clean up the resources by stopping `ProducerTemplate` and Camel 3. In the route, you use the `bean` method to invoke `HelloBean` by the `helloBean` name you gave it when it was enlisted in the registry 1.

You can try this test by going to the `chapter4/bean` directory and running this Maven goal:

```
mvn test -Dtest=SimpleRegistryTest
```

The next registry is for when you use Spring together with Camel.

4.3.3 APPLICATIONCONTEXTREGISTRY

`ApplicationContextRegistry` is the default registry when Camel is

used in a Spring environment such as Spring Boot or from a Spring XML file. When we say *using Camel in Spring XML*, we mean the following, as this snippet illustrates:

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
  <route>  
    <from uri="direct:start"/>  
    <bean ref="helloBean" method="hello"/>  
  </route>  
</camelContext>
```

Defining Camel by using the `<camelContext>` tag automatically lets Camel use `ApplicationContextRegistry`. This registry allows you to define beans in Spring XML files as you would normally do when using Spring. For example, you could define the `helloBean` bean as follows:

```
<bean id="helloBean" class="camelinaction.HelloBean"/>
```

It can hardly be simpler than that. When you use Camel with Spring, you can keep on using Spring beans as you would normally, and Camel will use those beans seamlessly without any configuration.

The next two registries apply when you use Camel with OSGi.

4.3.4 OSGISERVICEREGISTRY AND BLUEPRINTCONTAINERREGISTRY

When Camel is used in an OSGi environment, Camel uses a two-step lookup process. First, it looks up whether a service with the name exists in the OSGi service registry. If not, Camel falls back and looks up the name in `BlueprintContainerRegistry` when using OSGi Blueprint.

Popular OSGi platforms with Camel

The most popular OSGi platform to use with Apache Camel is Apache Karaf or Apache ServiceMix. You can also find

commercial platforms from vendors such as Red Hat and Talend.

Suppose you want to allow other bundles in OSGi to reuse the following bean:

```
<bean id="helloBean" class="camelaction.HelloBean"/>
```

You can do that by exporting the bean as an OSGi service that will enlist the bean into the OSGi service registry, as follows:

```
<osgi:service id="helloService"
interface="camelaction.HelloBean"
ref="helloBean"/>
```

Now another bundle such as a Camel application can reuse the bean, by referring to the service:

```
<osgi:reference id="helloService"
interface="camelaction.HelloBean"/>
```

To call the bean from the Camel route is easily done by using the bean component as if the bean is a local `<bean>` element in the same OSGi Blueprint XML file:

```
<camelContext
xmlns="http://camel.apache.org/schema/blueprint">
<route>
<from uri="direct:start"/>
<bean ref="helloService" method="hello"/>
</route>
</camelContext>
```

All you have to remember is the name with which the bean was exported. Camel will look it up in the OSGi service registry and the Blueprint bean container for you. This is convention over configuration.

The last registry is for using Camel with CDI.

4.3.5 CDIBEANREGISTRY

Contexts and Dependency Injection (CDI) is a Java specification

that standardizes how Java developers can integrate Java beans in a loosely coupled way. For Camel developers, it means you can use the CDI annotations to inject Java beans, Camel endpoints, and other services.

CDI containers

CDI is a Java specification that's part of Java EE, and therefore available in Java EE application servers such as Apache TomEE, WildFly, and in commercial offerings as well. But CDI is lightweight, and you can run in a standalone CDI container such as Apache OpenWebBeans or JBoss Weld. You'll encounter CDI again in chapters 7, 9, and 15 as you learn about CDI with microservices, testing, and running Camel. `cdiBeanRegistry` is the default registry when using Camel with CDI. The registry is automatically created in the default constructor of the `cdiCamelContext`, as shown here:

```
public class CdiCamelContext extends  
DefaultCamelContext {  
  
    public CdiCamelContext() {  
        super(new CdiBeanRegistry());  
  
        setInjector(new CdiInjector(getInjector()));  
    }  
}
```

Using `CdiBeanRegistry` as bean registry when using Camel with CDI—as a Camel end user, you don't need to configure this, because Camel with CDI is automatically configured

In CDI, beans can be defined by using CDI annotations. For example, to define a singleton bean with the name `helloBean`, you could do so as shown in the following listing.

Listing 4.6 A simple bean to print “Hello World” using CDI

annotations

@Singleton ①

①

The bean is singleton scoped

@Named("helloBean") ②

②

The bean is registered with the name helloBean

```
public class HelloBean {  
    private int counter;  
  
    public String hello() {  
        return "Hello " + ++counter + " times";  
    }  
}
```

To use the singleton bean ① from a Camel route, you can use a bean reference to look up the bean by its name ②. The Camel route could also be coded with CDI, as shown in the following listing.

[Listing 4.7](#) A Camel route using CDI to use the bean from listing 4.6

@Singleton ①

①

Defines the class as a CDI Singleton bean

```
public class HelloRoute extends RouteBuilder {  
  
    @EndpointInject(uri = "timer:foo?period=5s")  
    private Endpoint input; ②
```

②

Injects Camel endpoint that's used in the route

```
@EndpointInject(uri = "log:output")
private Endpoint output; 3
```

3

Injects Camel endpoint that's used in the route

```
@Override
public void configure() throws Exception {
    from(input)
        .bean("helloBean", "hello") 4
```

4

Invokes the bean by referring to the bean name

```
        .to(output);
    }
}
```

The route builder class has been annotated with `@Singleton` 1, which lets Camel automatically discover the route when starting. Then you inject endpoints by using `org.apache.camel.EndpointInject` 2 3. This isn't needed, because you could have used the endpoint URIs directly in the Java DSL route, which could have been written as follows:

```
from("timer:foo?period=5s")
    .bean("helloBean", "hello")
    .to("log:output");
```

But we've chosen to show a practice often used with CDI, which is dependency injection, and why we inject the endpoints also. To call the bean, you can use `bean("helloBean", "hello")` 4 to refer to the bean by its name and the name of the method to invoke. But you also could have injected the bean by using CDI. This can be done using `@Inject` and `@Named`, as shown in the following code snippet:

```
@Inject @Named("helloBean")
private HelloBean helloBean;

public void configure() throws Exception {
    from(input)
        .bean(helloBean)
        .to(output);
}
```

This example is provided in the book's source code in chapter4/cdi-beans. You can try the example with the following Maven goal:

```
mvn clean install camel:run
```

This concludes your tour of registries. Next we'll focus on how Camel selects which method to invoke on a given bean.

4.4 Selecting bean methods

You've seen how Camel works with beans from the route perspective. Now it's time to dig down and see the moving parts in action. You first need to understand the mechanism Camel uses to select the method to invoke.

Remember, Camel acts as a service activator using the bean component, which sits between the caller and the bean. At compile time, there are no direct bindings, and the JVM can't link the caller to the bean; Camel must resolve this at runtime.

Figure 4.4 illustrates how the bean component uses the registry to look up the bean to invoke.

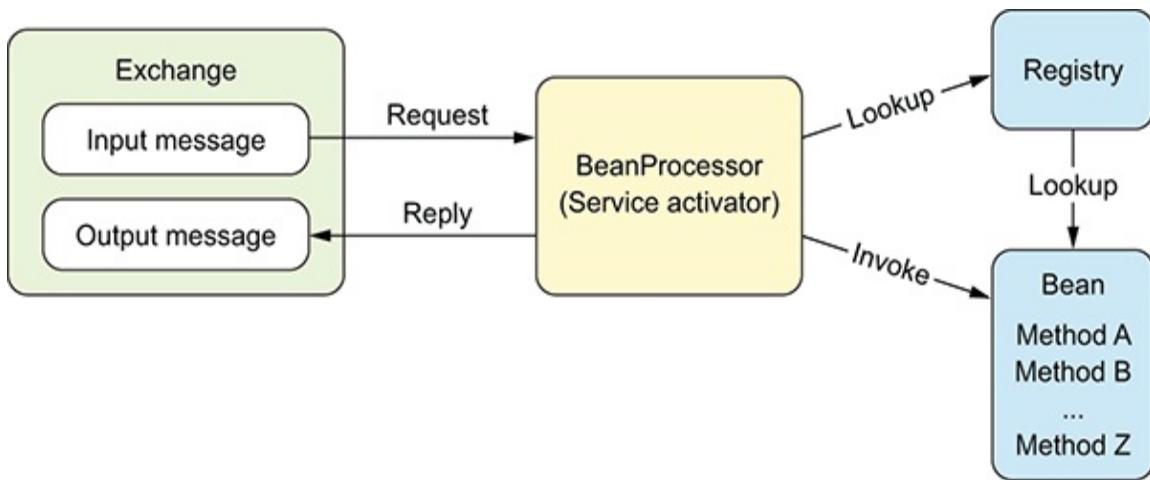


Figure 4.4 To invoke a bean in Camel, the bean component (BeanProcessor) looks it up in the registry, selects and adapts a method, invokes it, and passes the returned value as the reply to the Camel exchange.

At runtime, a Camel exchange is routed, and at a given point in the route, it reaches the bean component. The bean component (BeanProcessor) then processes the exchange, performing these general steps:

1. Looks up the bean in the registry
2. Selects the method to invoke on the bean
3. Binds to the parameters of the selected method (for example, using the body of the input message as a parameter; this is covered in detail in section 4.5)
4. Invokes the method
5. Handles any invocation errors that occur (any exceptions thrown from the bean will be set on the Camel exchange for further error handling)
6. Sets the method's reply (if there is one) as the body on the output message on the Camel exchange

Section 4.3 covered how registry lookups are done. The next two steps (steps 2 and 3 in the preceding list) are more complex, and we cover them in the remainder of this chapter. The reason this is more complex in Camel is that Camel has to compute which bean and method to invoke at runtime, whereas Java code is linked at compile time.

Why does Camel need to select a method?

Why is there more than one possible method name when you invoke a method? The answer is that beans can have overloaded methods, and in some cases the method name isn't specified either, which means Camel has to pick among all methods on the bean.

Suppose you have the following methods:

```
String echo(String s);
int echo(int number);
void doSomething(String something);
```

Camel has three methods to choose from. If you explicitly tell Camel to use the `echo` method, you're still left with two methods to choose from. We'll look at how Camel resolves this dilemma.

We'll first take a look at the algorithm Camel uses to select the method. Then we'll look at a couple of examples and see what could go wrong and how to avoid problems.

4.4.1 HOW CAMEL SELECTS BEAN METHODS

Unlike at compile time, when the Java compiler can link method invocations together, the bean component has to select the method to invoke at runtime.

Suppose you have the following class:

```
public class EchoBean {
    String echo(String name) {
        return name + " " + name;
    }
}
```

At compile time, you can express your code to invoke the `echo`

method like this:

```
EchoBean echo = new EchoBean();
String reply = echo.echo("Camel");
```

This ensures that the `echo` method is invoked at runtime. On the other hand, suppose you use the `EchoBean` in Camel in a route as follows:

```
from("direct:start")
    .bean(EchoBean.class, "echo")
    .to("log:reply");
```

When the compiler compiles this code, it can't see that you want to invoke the `echo` method on the `EchoBean`. From the compiler's point of view, `EchoBean.class` and `echo` are parameters to the bean method. All the compiler can check is that the `EchoBean` class exists; if you misspelled the method name, perhaps typing `ekko`, the compiler couldn't catch this mistake. The mistake would end up being caught at runtime, when the bean component would throw a `MethodNotFoundException` stating that the method named `ekko` doesn't exist.

Camel also allows you not to explicitly name a method. For example, you could write the previous route as follows:

```
from("direct:start")
    .bean(EchoBean.class)
    .to("log:reply");
```

Regardless of whether the method name is explicitly given, Camel has to compute which method to invoke. Let's look at how Camel chooses.

4.4.2 CAMEL'S METHOD-SELECTION ALGORITHM

The bean component uses a complex algorithm to select which method to invoke on a bean. You don't need to understand or remember every step in this algorithm; we simply want to outline what goes on inside Camel to make working with beans as simple as possible for you.

Figure 4.5 shows the first part of this algorithm, which is continued in figure 4.6.

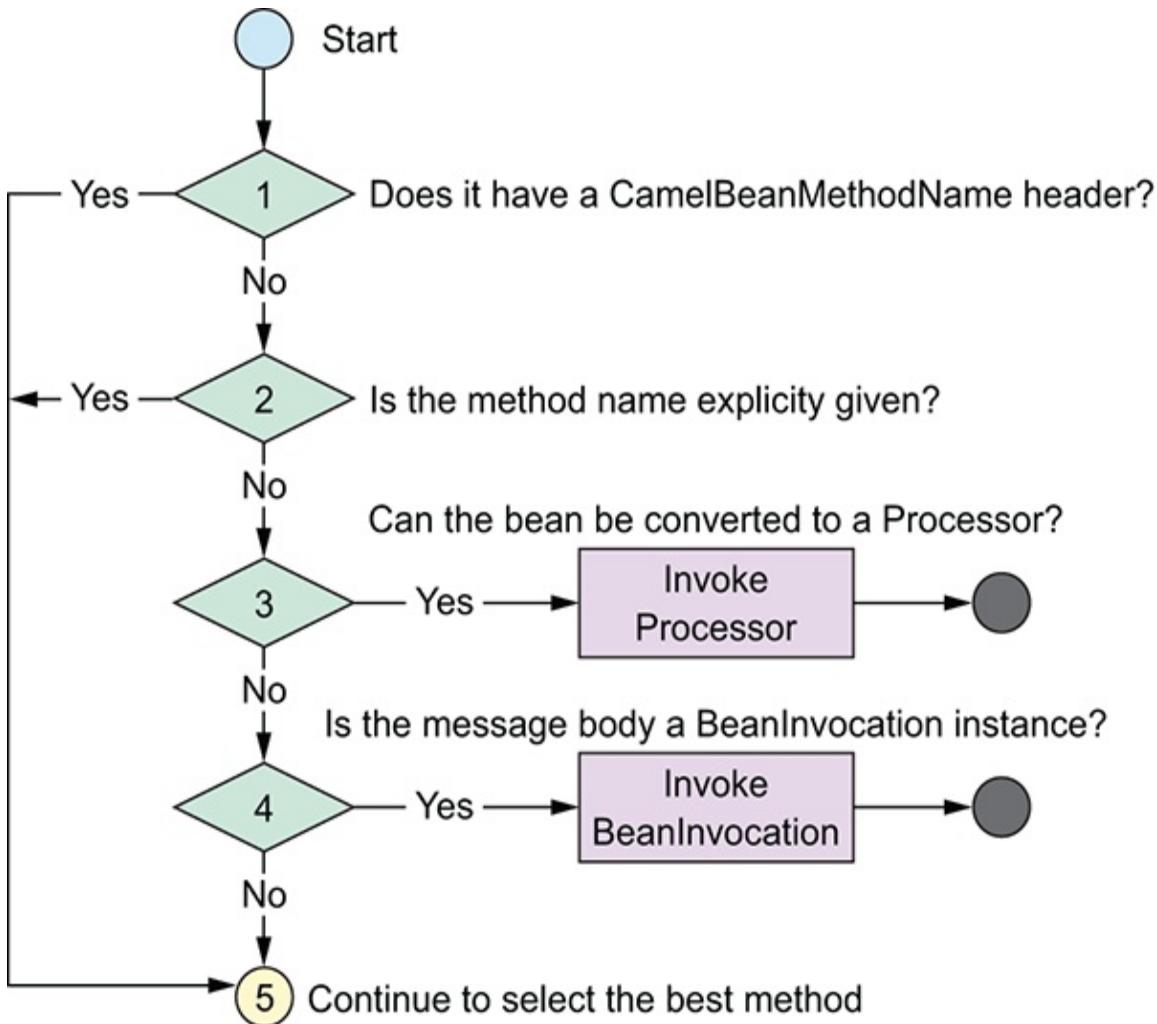


Figure 4.5 How Camel selects which method to invoke (part 1, continued in figure 4.6)

Here's how the algorithm selects the method to invoke:

1. If the Camel message contains a header with the key CamelBeanMethodName, its value is used as the explicit method name. Go to step 5.
2. If a method is explicitly defined, Camel uses it, as we mentioned at the start of this section. Go to step 5.
3. If the bean can be converted to a Processor using the Camel type-converter mechanism, the Processor is used to process the

message. This may seem odd, but it allows Camel to turn any bean into a message-driven bean equivalent. For example, with this technique, Camel allows any `javax.jms.MessageListener` bean to be invoked directly by Camel without any integration glue. This method is rarely used by end users of Camel, but it can be a useful trick.

4. If the body of the Camel message can be converted into `org.apache.camel.component.bean.BeanInvocation`, that's used to invoke the method and pass the arguments to the bean. This is in use only when using the Camel bean proxy, which is rarely used and not covered in this book. (You can find coverage of this subject in the first edition of the book in chapter 14.)
5. Continue with the second part of the algorithm, shown in [figure 4.6.](#)

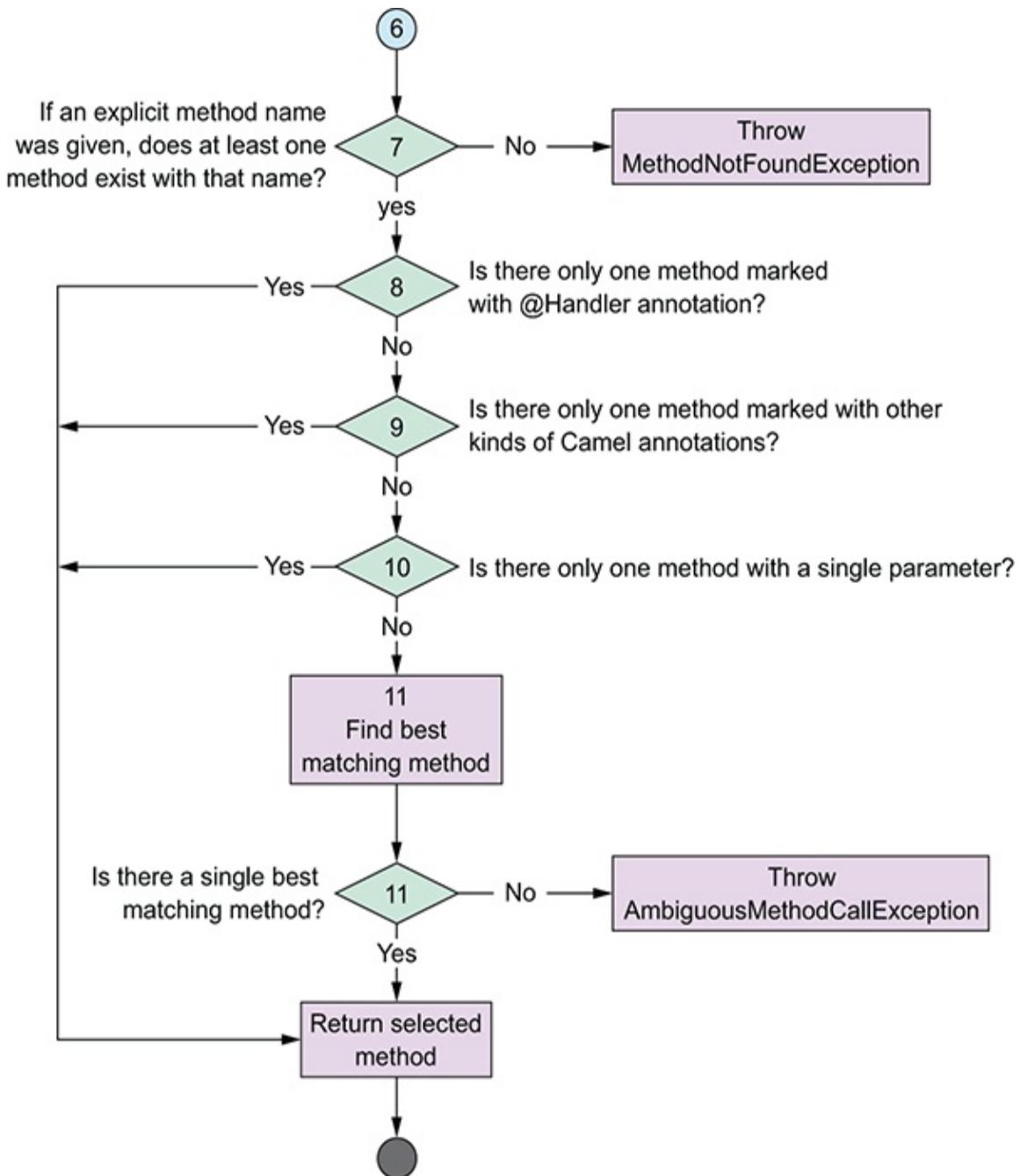


Figure 4.6 How Camel selects which method to invoke (part 2, continued from figure 4.5)

Figure 4.6 is a bit more complex, but its main goal is to narrow the number of possible methods and select a method if one stands out. Don't worry if you don't entirely understand the algorithm; you'll look at a couple of examples shortly that should make it much clearer.

Let's continue with the algorithm and cover the last steps:

1. If a method name was given and no methods exist with that name, a `MethodNotFoundException` exception is thrown.
2. If only a single method has been marked with the `@Handler` annotation, it's selected.
3. If only a single method uses any of the other Camel bean parameter-binding annotations, such as `@Body`, `@Header`, and so on, it's selected. (You'll look at how Camel binds to method parameters using annotations in section 4.4.3.)
4. If, among all the methods on the bean, there's only one method with exactly one parameter, that method is selected. For example, this would be the situation for the `EchoBean` bean you looked at in section 4.4.1, which has only the `echo` method with exactly one parameter. Single-parameter methods are preferred because they map easily with the payload from the Camel exchange.
5. Now the computation gets a bit complex. There are multiple candidate methods, and Camel must determine whether a single method stands out as the best fit. The strategy is to go over the candidate methods and filter out methods that don't fit. Camel does this by trying to match the first parameter of the candidate method; if the parameter isn't the same type and it's not possible to coerce the types, the method is filtered out. If a method name includes numerous parameter values, these values are also used during filtering. The values are matched with the pairing parameter type on the candidate methods. In the end, if only a single method is left, that method is selected. Because this logic is elaborate, we cover it in more detail in the following sections.
6. If Camel can't select a method, an `AmbiguousMethodCallException` exception is thrown with a list of ambiguous methods.

Clearly, Camel goes through a lot to select the method to invoke on your bean. Over time you'll learn to appreciate all this—it's convention over configuration to the fullest.

NOTE The algorithm laid out in this book is based on Apache Camel version 2.20. This method-selection algorithm may change in the future to accommodate new features.

Now it's time to look at applying this algorithm in practice.

4.4.3 SOME METHOD-SELECTION EXAMPLES

To see how this algorithm works, you'll use the EchoBean from section 4.4.1 as an example. This time, you'll add another method to it—the bar method—to better understand what happens when you have multiple candidate methods:

```
public class EchoBean {  
  
    public String echo(String echo) {  
        return echo + " " + echo;  
    }  
  
    public String bar() {  
        return "bar";  
    }  
}
```

And you'll start with this route:

```
from("direct:start")  
    .bean(EchoBean.class)  
    .to("log:reply");
```

If you send the string message camel to the Camel route, the reply logger will surely output camel camel as expected, since the EchoBean will duplicate the input as its response. Although EchoBean has two methods, echo and bar, only the echo method has a single parameter. This is what step 9 in figure 4.6 ensures: Camel will pick the method with a single parameter if there's only one of them.

To make the example more challenging, let's change the bar method as follows:

```
public String bar(String name) {
```

```
    return "bar " + name;  
}
```

What do you expect will happen now? You have two identical method signatures with a single method parameter. In this case, Camel can't pick one over the other, so it throws an `AmbiguousMethodCallException` exception, according to step 11 in [figure 4.6](#).

How can you resolve this? One solution is to provide the method name in the route, such as specifying the `bar` method:

```
from("direct:start")  
    .bean(EchoBean.class, "bar")  
    .to("log:reply");
```

But there's another solution that doesn't involve specifying the method name in the route: you can use the `@Handler` annotation to select the method. This solution is implemented in step 7 of [figure 4.6](#). `@Handler` is a Camel-specific annotation that you can add to a method. It simply tells Camel to use this method by default:

```
@Handler  
public String bar(String name) {  
    return "bar " + name;  
}
```

Now `AmbiguousMethodCallException` won't be thrown because the `@Handler` annotation tells Camel to select the `bar` method.

TIP It's a good idea either to declare the method name in the route or to use the `@Handler` annotation. This ensures that Camel picks the method you want, and you won't be surprised if Camel chooses another method.

Suppose you change `EchoBean` to include two methods with different parameter types:

```
public class EchoBean {
```

```

public String echo(String echo) {
    return echo + " " + echo;
}

public Integer doubleUp(Integer num) {
    return num.intValue() * num.intValue();
}
}

```

The `echo` method works with a `String`, and the `doubleUp` method with an `Integer`. If you don't specify the method name, the bean component will have to choose between these two methods at runtime.

Step 10 in [figure 4.6](#) allows Camel to be smart about deciding which method stands out. It does so by inspecting the message payloads of two or more candidate methods and comparing those with the message body type, checking whether there's an exact type match in any of the methods.

Suppose you send a message to the route that contains a `String` body with the word `camel`. It's not hard to guess that Camel will pick the `echo` method, because it works with `String`. On the other hand, if you send in a message with the `Integer` value of 5, Camel will select the `doubleup` method, because it uses the `Integer` type.

Despite this, things can still go wrong, so let's go over a couple common situations.

4.4.4 POTENTIAL METHOD-SELECTION PROBLEMS

A few things can go wrong when invoking beans at runtime:

- *Specified method not found*—If Camel can't find any method with the specified name, a `MethodNotFoundException` exception is thrown. This happens only when you've explicitly specified the method name.
- *Ambiguous method*—If Camel can't single out a method to call, an `AmbiguousMethodCallException` exception is thrown with a list

of the ambiguous methods. This can happen even when an explicit method name is defined, because the method could potentially be overloaded, which means the bean would have multiple methods with the same name; only the number of parameters would vary.

- *Type conversion failure*—Before Camel invokes the selected method, it must convert the message payload to the parameter type required by the method. If this fails, a `NoTypeConversionAvailableException` exception is thrown.

Let's take a look at examples of each of these three situations using the following EchoBean:

```
public class EchoBean {  
  
    public String echo(String name) {  
        return name + name;  
    }  
  
    public String hello(String name) {  
        return "Hello " + name;  
    }  
}
```

First, you could specify a method that doesn't exist by doing this:

```
.bean("echoBean", "foo")
```

And in XML DSL:

```
<bean ref="echoBean" method="foo"/>
```

Here you try to invoke the `foo` method, but no such method exists, so Camel throws a `MethodNotFoundException` exception.

On the other hand, you could omit specifying the method name:

```
.bean("echoBean")
```

And in XML DSL:

```
<bean ref="echoBean"/>
```

In this case, Camel can't single out a method to use because both the `echo` and `hello` methods are ambiguous. When this happens, Camel throws an `AmbiguousMethodCallException` exception containing a list of the ambiguous methods.

The last situation that could happen is when the message contains a body that can't be converted to the type required by the method. Suppose you have the following `OrderServiceBean` class:

```
public class OrderServiceBean {  
    public String handleXML(Document xml) {  
        ...  
    }  
}
```

And suppose you need to use that bean in this route:

```
from("jms:queue:orders")  
    .bean("orderService", "handleXML")  
    .to("jms:queue:handledOrders");
```

The `handleXML` method requires a parameter to be of type `org.w3c.dom.Document`, which is an XML type, but what if the JMS queue contains a `javax.jms.TextMessage` not containing any XML data, but just a plain-text message, such as `camel rocks`? At runtime, you'll get the following stack trace:

```
org.apache.camel.TypeConversionException: Error during type  
conversion from  
type: java.lang.String to the required type:  
org.w3c.dom.Document with value  
Camel rocks due org.xml.sax.SAXParseException; lineNumber:  
1; columnNumber:  
1; Content is not allowed in prolog.  
    at  
org.apache.camel.impl.converter.BaseTypeConverterRegistry.c  
reateType  
ConversionException(BaseTypeConverterRegistry.java:571)  
    at  
org.apache.camel.impl.converter.BaseTypeConverterRegistry.c  
onvertTo  
(BaseTypeConverterRegistry.java:129)  
    at  
org.apache.camel.impl.converter.BaseTypeConverterRegistry.c
```

```
onvertTo  
(BaseTypeConverterRegistry.java:100)  
Caused by: org.xml.sax.SAXParseException; lineNumber: 1;  
columnNumber:  
1;Content is not allowed in prolog.  
at  
com.sun.org.apache.xerces.internal.parsers.DOMParser.parse  
(DOMParser.java:257)  
at  
com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderImpl  
.parse  
(DocumentBuilderImpl.java:348)  
at  
org.apache.camel.converter.jaxp.XmlConverter.toDOMDocument  
(XmlConverter.java:902)
```

Camel tries to convert the `javax.jms.TextMessage` to an `org.w3c.dom.Document` type, but it fails. In this situation, Camel wraps the error and throws it as a `TypeConversionException` exception.

By looking further into this stack trace, you may notice that the cause of this problem is that the XML parser couldn't parse the data to XML. It reports `Content is not allowed in prolog`, which is a common error indicating that the XML declaration (`<?xml version="1.0"?>`) is missing, and therefore a strong indicator that the payload isn't XML based.

NOTE You may wonder what would happen if such a situation occurred. In this case, the Camel error-handling system would kick in and handle it. Error handling is covered thoroughly in chapter 11.

Before we wrap up this section, we'd like you to know about one more functionality. It covers use cases in which your beans have overloaded methods (methods using the same name, but with different parameter types, for example) and how Camel works in those situations.

4.4.5 METHOD SELECTION USING TYPE

MATCHING

The previous algorithm in step 10 of figure 4.6 also allows users to filter methods based on parameter-type matching. For example, suppose the following class has multiple methods to handle the order:

```
public class OrderServiceBean {  
  
    public String handleXML(Document xml) {  
        ...  
    }  
  
    public String handleXML(String xml) {  
        ...  
    }  
}
```

And you need to use that bean in this route:

```
from("jms:queue:orders")  
    .bean("orderService", "handleXML")  
    .to("jms:queue:handledOrders");
```

OrderServiceBean has two methods named `handleXML`, each one accepting a different parameter type as the input. What happens at runtime depends on whether the bean component is able to find a single suitable method to invoke, according to the algorithm listed in figures 4.5 and 4.6. To decide that Camel will be based on the message body class type, determine which one is the best candidate (if possible). If the message body is, for example, a `Document` or `String` type, there's a direct match with the types on those methods and the appropriate method is invoked. But if the Camel message body is of any other type, such as `java.io.File`, Camel will look up whether there are type converters that can convert from `java.io.File` and respectively to `org.w3c.Document` and `java.lang.String`. If only one type conversion is possible, the appropriate method is chosen. In any other case, `AmbiguousMethodCallException` is thrown.

In those situations, you can assist Camel by explicitly defining which of the two methods to call by specifying which type to use,

as shown here:

```
from("jms:queue:orders")
    .bean("orderService", "handleXML(org.w3c.Document)")
    .to("jms:queue:handledOrders");
```

You'd need to specify the class type by using its fully qualified name. But for common types such as Boolean, Integer, and String, you can omit the package name, and use just String as the java.lang.String type:

```
from("jms:queue:orders")
    .bean("orderService", "handleXML(String)")
    .to("jms:queue:handledOrders");
```

Over the years with Camel, we haven't often seen the need for this. If possible, we suggest using unique method names instead, which makes it easier for both Camel and end users to know exactly which methods Camel will use.

That's all you need to know about how Camel selects methods at runtime. Now you need to look at the bean parameter-binding process, which happens after Camel has selected the method.

4.5 Performing bean parameter binding

The preceding section covered the process that selects which method to invoke on a bean. This section covers what happens next—how Camel adapts to the parameters on the method signature. Any bean method can have multiple parameters, and Camel must somehow pass in meaningful values. This process is known as *bean parameter binding*.

You've already seen parameter binding in action in the many examples so far in this chapter. What those examples have in common is using a single parameter to which Camel bound the input message body. [Figure 4.7](#) illustrates this, using EchoBean as an example.

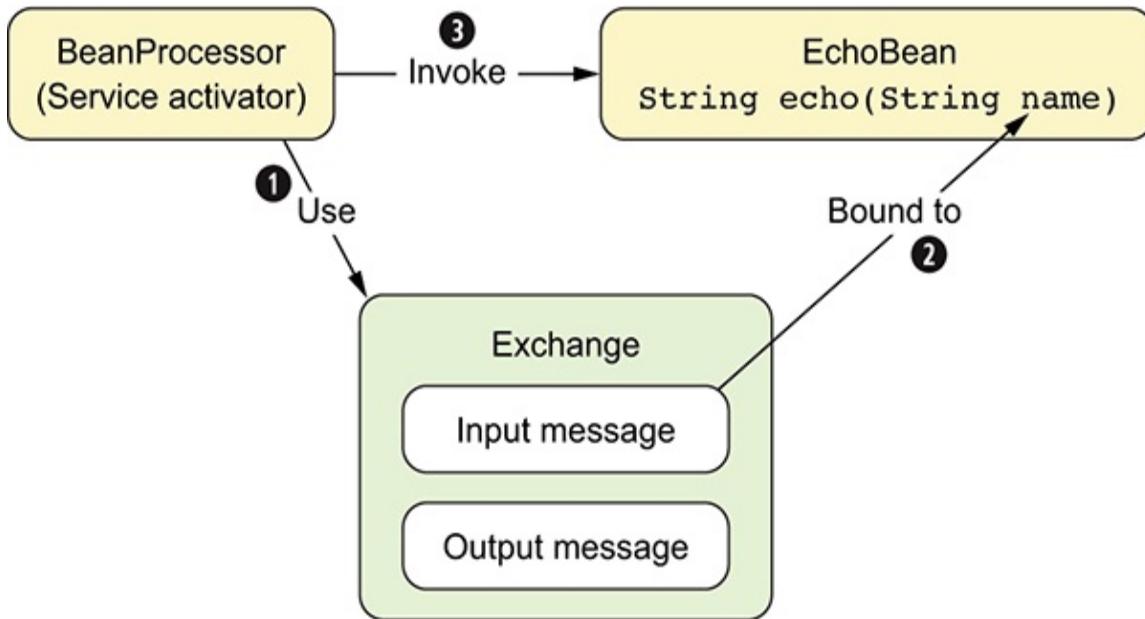


Figure 4.7 How a bean component binds the input message to the first parameter of the method being invoked

The bean component (BeanProcessor) uses the input message ① to bind its body to the first parameter of the method ②, which happens to be the `String name` parameter. Camel does this by creating an expression that type-converts the input message body to the `String` type. This ensures that when Camel invokes the `echo` method ③, the parameter matches the expected type.

This is important to understand, because most beans have methods with a single parameter. The first parameter is expected to be the input message body, and Camel will automatically convert the body to the same type as the parameter.

What happens when a method has multiple parameters? That's what we'll look at in the remainder of this section.

4.5.1 BINDING WITH MULTIPLE PARAMETERS

Figure 4.8 illustrates the principle of bean parameter binding when multiple parameters are used.

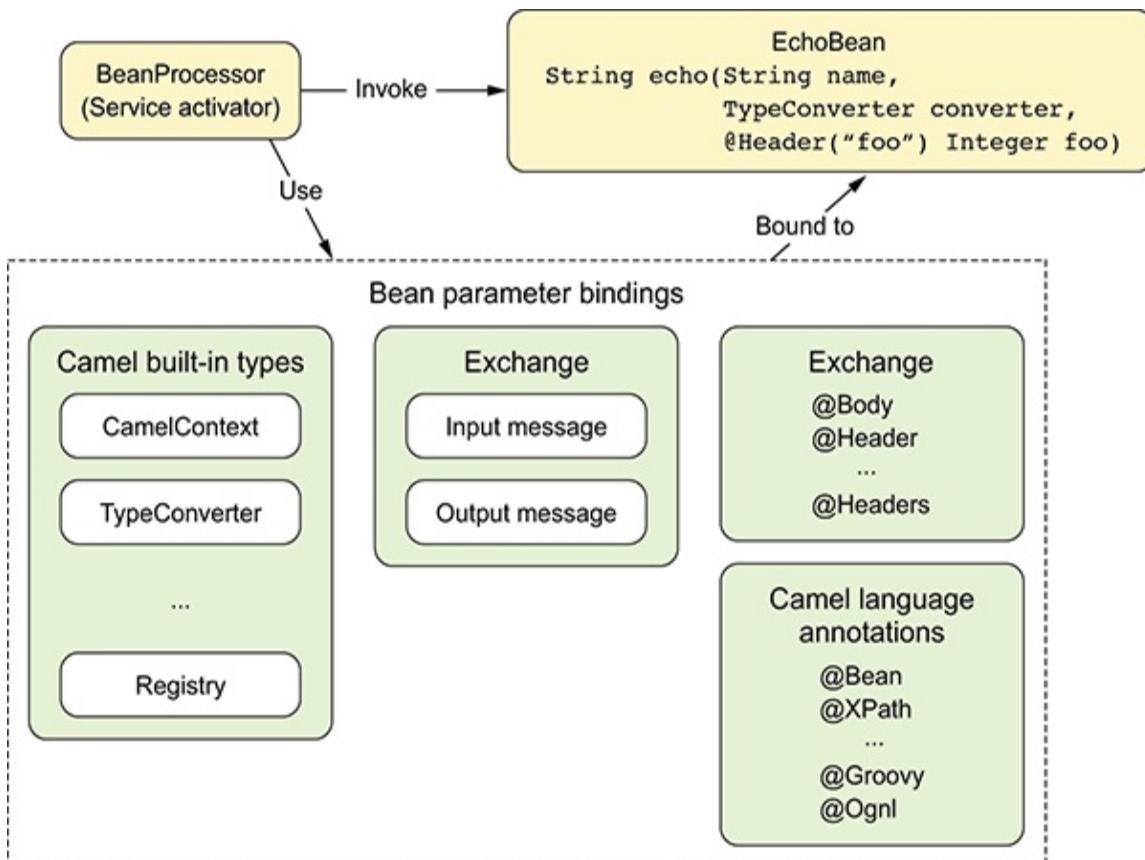


Figure 4.8 Parameter binding with multiple parameters involves a lot more options than with single parameters.

At first, [figure 4.8](#) may seem overwhelming. Many new types come into play when you deal with multiple parameters. The big box entitled *Bean parameter bindings* contains the following four boxes:

- *Camel built-in types*—Camel provides special bindings for a series of Camel concepts. We cover them in section 4.5.2.
- *Exchange*—This is the Camel exchange, which allows binding to the input message, such as its body and headers. The Camel exchange is the source of the values that must be bound to the method parameters. It's covered in the sections to come.
- *Camel annotations*—When dealing with multiple parameters, you use annotations to distinguish them. This is covered in section 4.5.3.
- *Camel language annotations*—This is a less commonly used

feature that allows you to bind parameters to languages. It's ideal when working with XML messages that allow you to bind parameters to XPath expressions. This is covered in section 4.5.4.

In addition, you can specify the parameter binding by using the method-name signature that resembles how you call methods in Java source code. We'll look at this in section 4.5.5.

Working with multiple parameters

Using multiple parameters is more complex than using single parameters. It's generally a good idea to follow these rules of thumb:

- Use the first parameter for the message body, which may or may not use the `@Body` annotation.
- Use either a built-in type or add Camel annotations for subsequent parameters.
- When having more than two parameters, consider specifying the binding in the method-name signature, which makes it clear for humans and Camel how each parameter should be mapped. We cover this in section 4.5.5.

In our experience, it becomes complicated when multiple parameters don't follow these guidelines, but Camel will make its best attempt to adapt the parameters to the method signature.

Let's start by looking at using the Camel built-in types.

4.5.2 BINDING USING BUILT-IN TYPES

Camel provides a set of fixed types that are always bound. All you have to do is declare a parameter of one of the types listed in table 4.2.

Table 4.2 Parameter types that Camel automatically binds

Type	Description
E x c h a n g e	The Camel exchange. This contains the values that will be bound to the method parameters.
M e s s a g e	The Camel input message. It contains the body that's often bound to the first method parameter.
c a m e l c o n t e xt	The Camel context. This can be used in special circumstances when you need access to all of Camel's moving parts.
T y p e C o n v e r t er	The Camel type-converter mechanism. This can be used when you need to convert types. Chapter 3 covered the type-converter mechanism.
R e g i s t ry	The bean registry. This allows you to look up beans in the registry.
E x c	An exception, if one was thrown. Camel will bind to this only if the exchange has failed and contains an exception. This allows you to use beans to handle errors. Chapter 11 covers error handling; you can find an example of using a

e
p
t
i
on

custom bean to handle failures in section 11.4.4.

Let's look at a couple of examples using the types from table [4.2](#). First, suppose you add a second parameter that's one of the built-in types to the echo method:

```
public string echo(String echo, CamelContext context)
```

In this example, you bind `CamelContext`, which gives you access to all the moving parts of Camel.

Or you could bind the registry, in case you need to look up some beans:

```
public string echo(String echo, Registry registry) {  
    OtherBean other = registry.lookup("other",  
    OtherBean.class);  
    ...  
}
```

You aren't restricted to having only one additional parameter; you can have as many as you like. For example, you could bind both the `CamelContext` and the registry:

```
public string echo(String echo, CamelContext context,  
Registry registry)
```

So far, you've always bound to the message body. How would you bind to a message header? The next section explains that.

4.5.3 BINDING USING CAMEL ANNOTATIONS

Camel provides a range of annotations to help bind from the exchange to bean parameters. You should use these annotations when you want more control over the bindings. For example, without these annotations, Camel will always try to bind the method body to the first parameter, but with the `@Body` annotation, you can bind the body to any parameter in the method.

Suppose you have the following bean method:

```
public String orderStatus(Integer customerId, Integer  
orderId)
```

And you have a Camel message that contains the following data:

- Body, with the order ID, as a `String` type
- Header with the customer ID as an `Integer` type

With the help of Camel annotations, you can bind the exchange to the method signature as follows:

```
public String orderStatus(@Header("customerId") Integer  
customerId,  
                           @Body Integer orderId)
```

Notice that you can use the `@Header` annotation to bind the message header to the first parameter and `@Body` to bind the message body to the second parameter.

Table 4.3 lists all the Camel parameter-binding annotations.

Table 4.3 Camel parameter-binding annotations

Annotation	Description
<code>@Body</code>	Binds the parameter to the message body.
<code>@Header(name)</code>	Binds the parameter to the message header with the given name.
<code>@Headers</code>	Binds the parameter to all the input headers. The parameter must be a <code>java.util.Map</code> type.
<code>@ExchangeProperty(name)</code>	Binds the parameter to the exchange property with the given name.
<code>@ExchangeProperties</code>	Binds the parameter to all the exchange properties. The parameter must be a <code>java.util.Map</code> type.
<code>@ExchangeException</code>	Binds the parameter to the exception set on the exchange.
<code>@Attachments</code>	Binds the parameter to the message attachments. The parameter must be a <code>java.util.Map</code> type.

You've already seen the first type in action, so let's try a couple examples with the other annotations. For example, you could use `@Header` to bind a named header to a parameter, and this can be done more than once, as shown here:

```
public String orderStatus(@Body Integer orderId,
                           @Header("customerId") Integer
                           customerId,
                           @Header("customerType") Integer
                           customerType) {
    ...
}
```

If you have many headers, it may be easier to use `@Headers` to bind all the headers to a `Map` type:

```
public String orderStatus(@Body Integer orderId, @Headers
                           Map headers) {
    Integer customerId = (Integer)
                           headers.get("customerId");
    String customerType = (String)
                           headers.get("customerType");
    ...
}
```

Finally, let's look at Camel's language annotations, which bind parameters to a language.

4.5.4 BINDING USING CAMEL LANGUAGE ANNOTATIONS

Camel provides additional annotations that allow you to use other languages as parameters. One of the most common languages to use is XPath, which allows you to evaluate XPath expressions on the message body as XML documents. For example, suppose the message contains the following XML document:

```
<order customerId="123">
    <status>in progress</status>
</order>
```

By using XPath expressions, you can extract parts of the

document and bind them to parameters, like this:

```
public void updateStatus(@XPath("/order/@customerId")
Integer customerId,
@XPath("/order/status/text()") String status)
```

You can bind as many parameters as you like—the preceding example binds two parameters by using the @XPath annotations. You can also mix and match annotations, so you can use @XPath for one parameter and @Header for another.

Table 4.4 lists the most commonly used language annotations provided in Camel.

Table 4.4 Camel’s language-based bean binding annotations

Annotation	Description	Maven dependency
@Bean	Invokes a method on a bean	camel-core
@Constant	Evaluates as a constant value	camel-core
@Groovy	Evaluates a Groovy script	camel-script
@JavaScript	Evaluates a JavaScript script	camel-script
@JsonPath	Evaluates a JsonPath expression	camel-jsonpath
@MVEL	Evaluates a MVEL script	camel-mvel
@Simple	Evaluates a Simple expression. (Simple is a built-in language provided with Camel; see appendix A for more details.)	camel-core
@SpEL	Evaluates a Spring expression	camel-spring
@XPath	Evaluates an XPath expression	camel-core

@XQuery	Evaluates an XQuery expression	camel-saxon
---------	--------------------------------	-------------

It may seem a bit magical that you can use an @Bean annotation when invoking a method, because the @Bean annotation itself also invokes a method. Let's try an example.

Suppose you already have a service that must be used to stamp unique order IDs on incoming orders. The service is implemented in the following listing.

Listing 4.8 A service that stamps an order ID on an XML document

```
public Document handleIncomingOrder(Document xml, int
customerId,
                                     int orderId) {
    Attr attr = xml.createAttribute("orderId"); ❶
```

❶

Creates orderId attribute

```
attr.setValue("") + orderId);
Node node = xml.getElementsByTagName("order").item(0);
node.getAttributes().setNamedItem(attr); ❷
```

❷

Adds orderId attribute to order node

```
return xml;
}
```

As you can see, the service creates a new XML attribute with the value of the given order ID ❶. Then it inserts this attribute in the XML document ❷ using the rather clumsy XML API from Java ❷.

To generate the unique order ID, you have the following class:

```
public final class GuidGenerator {
```

```

public static int generate() {
    Random ran = new Random();
    return ran.nextInt(10000000);
}
}

```

(In a real system, you'd generate unique order IDs based on another scheme.)

In Camel, you have the following route that listens for new order files and invokes the service before sending the orders to a JMS destination for further processing:

```

<bean id="xmlOrderService"
      class="camelaction.XmlOrderService"/>

<camelContext id="camel"
      xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file://riderautoparts/order/inbox"/>
        <bean ref="xmlOrderService"/>
        <to uri="jms:queue:order"/>
    </route>
</camelContext>

```

What's missing is the step that generates a unique ID and provides that ID in the `handleIncomingOrder` method (shown in [listing 4.8](#)). To do this, you need to declare a bean in the Spring XML file with the ID generator, as follows:

```
<bean id="guid" class="camelaction.GuidGenerator"/>
```

Now you're ready to connect the last pieces of the puzzle. You need to tell Camel that it should invoke the `generate` method on the `guid` bean when it invokes the `handleIncomingOrder` method from [listing 4.8](#). To do this, you use the `@Bean` annotation and change the method signature (highlighted in bold) to the following:

```

public Document handleIncomingOrder(@Body Document xml,
                                    @XPath("/order/@customerId") int
customerId,
                                    @Bean(ref = "guid", method="generate")
int orderId);

```

We've prepared a unit test you can use to run this example. Use the following Maven goal from the chapter4/bean directory:

```
mvn test -Dtest=XmlOrderTest
```

When it's running, you should see two log lines that output the XML order before and after the service has stamped the order ID. Here's an example:

```
2017-05-17 16:18:58,485 [: FileComponent] INFO before
Exchange[BodyType:org.apache.camel.component.file.GenericFile,
Body:<order customerId="4444"><item>Camel in action</item>
</order>]
2017-05-17 16:18:58,564 [: FileComponent] INFO after
Exchange[BodyType:com.sun.org.apache.xerces.internal.dom.
DeferredDocumentImpl, Body:<order customerId="4444"
orderId="7303381"><item>Camel in action</item></order>]
```

Here you can see that the second log line has an `orderId` attribute with the value of 7303381, whereas the first doesn't. If you run it again, you'll see a different order ID because it's a random value. You can experiment with this example, perhaps changing how the order ID is generated.

USING NAMESPACES WITH @XPATH

In the preceding example, the XML order didn't include a namespace. When using namespaces, the bean parameter binding must include the namespace(s) in the method signature as highlighted:

```
public Document handleIncomingOrder(
    @Body Document xml,
    @XPath(
        value = "/c:order/@customerId",
        namespaces = @NamespacePrefix(
            prefix = "c",
            uri = "http://camelinaction.com/order")) int
customerId,
    @Bean(ref = "guid", method = "generate") int orderId);
```

The namespace is defined by using the `@NamespacePrefix`

annotation embedded in the `@XPath` annotation. Notice that the XPath expression value must use the prefix, which means the expression is changed from `/order/@customerId` to `/c:order/@customerId`.

In recent times, JSON has become increasing more popular to use as a data format when exchanging data. Now imagine that the previous example uses JSON instead of XML; let's see how the Camel `@JsonPath` binding annotation works in practice.

USING `@JsonPath` BINDING ANNOTATION

To use `@JsonPath`, you first have to include the Camel component, which Maven users can do by adding the following dependency to their `pom.xml` file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jsonpath</artifactId>
    <version>2.20.1</version>
</dependency>
```

Instead of dealing with incoming orders as XML documents, the orders are now in JSON format, as in the sample shown here:

```
{
    "order": {
        "customerId": 4444,
        "item": "Camel in Action"
    }
}
```

The following listing shows how to transform incoming orders in JSON format to a CSV representation ② by mapping the `customerId` and `item` fields ① to bean parameters by using the `@JsonPath` annotation.

Listing 4.9 A Service that transforms JSON to CSV by using a Java bean

```
import org.apache.camel.jsonpath.JsonPath;
import org.apache.camel.language.Bean;
```

```
public class JsonOrderService {  
    public String handleIncomingOrder(  
        @JsonPath("$.order.customerId") int  
customerId, ①
```

①

Bindings from JSON document to bean parameters using @JsonPath

```
    @JsonPath("$.order.item") String item,  
    @Bean(ref = "guid", method = "generate")  
int orderId) {  
  
    return String.format("%d,%d,%s", orderId, customerId,  
item); ②
```

②

Returns a CSV representation of the incoming data

```
}
```

The book's source code contains this example in the chapter4/json directory. Maven users can run the example by using the following Maven goal:

```
mvn test -Dtest=JsonOrderTest
```

When running the example, you should see a log after the message transformation, such as this:

```
INFO after - Exchange[ExchangePattern: InOnly, BodyType:  
String, Body:5619507,4444,'Camel in Action']
```

TIP JsonPath allows you to work with JSON documents as XPath does for XML. As such, JsonPath offers a syntax that allows you to define expressions and predicates. For more information about the syntax, consult the JsonPath documentation at <https://github.com/json-path/JsonPath>.

What you've seen in this and the previous section is the need to use Camel annotations to declare the required binding information. Using Java annotations is common practice, but the caveat is that adding these annotations requires you to alter the source code of the bean. What if you could specify the binding without having to change the source code of the bean? The following section explains how this is possible.

4.5.5 PARAMETER BINDING USING METHOD NAME WITH SIGNATURE

Camel also allows you to specify the parameter-binding information by using a syntax that's similar to calling methods in Java. This requires using the method-name header, including the binding information as the method signature.

Pros and cons

This is a powerful technique, as it completely decouples Camel from your Java bean. Your Java bean can stay as *is* without having to import any Camel code or dependencies at compile time nor runtime. The caveat is that the binding must be defined in a string value that prevents any compile-time checking. In addition, the binding information in the string is limited to what the Simple language provides. For example, the `@JsonPath` language annotation covered in the previous section isn't available in the Simple language.

It's easier to explain with an example.

Suppose you have this method used previously as an example in section 4.5.3:

```
public String orderStatus(@Body Integer orderId,  
                           @Header("customerId") Integer  
                           customerId,
```

```
        @Header("customerType") Integer  
customerType) {  
    ...  
}
```

Instead of using the Camel annotations, the source code becomes this:

```
public String orderStatus(Integer orderId,  
                           Integer customerId,  
                           Integer customerType) {  
    ...  
}
```

Now the method is clean and has no Camel annotations, and the code has no Camel dependencies. The code can compile without having Camel JARs on the classpath. What you have to do is to specify the binding details in the Camel route instead:

```
from("direct:start")  
    .bean("orderService",  
          "orderStatus(${body}, ${header.customerId},  
${header.customerType});
```

And in XML DSL:

```
<route>  
    <from uri="direct:start"/>  
    <bean ref="orderService" method="  
        orderStatus(${body}, ${header.customerId},  
${header.customerType})"/>  
</route>
```

What happened? If you take a closer look, you'll see that the method-name parameter almost resembles Java source code, as if calling a method with three parameters. If you were to write some Java code and use the Camel Exchange API to call the method from Java, the source code would be something like this:

```
OrderStatus bean = ...  
Message msg = exchange.getIn();  
String status = bean.orderStatus(msg.getBody(),  
msg.getHeader("customerId",  
  
msg.getHeader("customerType"));
```

This code would be Java source code and therefore compiled by the Java compiler, so the parameter binding happens at compile time—whereas the preceding Camel routes are defined using a String value in Java code, or an XML attribute. Because it's not the Java compiler that performs the binding, Camel parses the String value, which happens upon starting up Camel. [Figure 4.9](#) illustrates how each of the three Java parameters corresponds to a value.

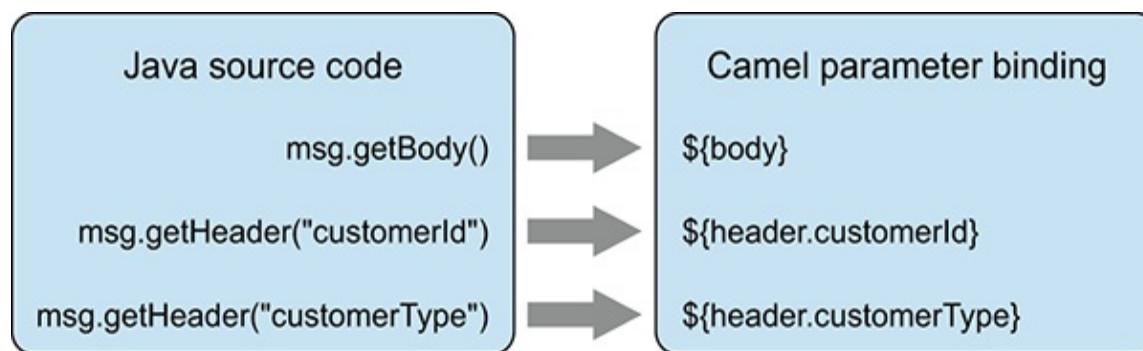


Figure 4.9 Bean parameter bindings in Camel resemble Java source code when calling a parameter with three values.

You may have guessed what syntax the Camel parameter binding is using, from the right-hand side in [figure 4.9](#). Yes, it's the Simple language. Camel allows you to specify the bean parameter binding by using the method-name signature. [Table 4.5](#) lists the rules that apply when using parameter binding and the method-name signature.

Table 4.5 Rules for bean parameter binding using method-name signature

Rule	Description
A Boolean value	The parameter can bind to a Boolean type by using a <code>true</code> or <code>false</code> value. For example: <code>method="goldCustomer(true)"</code> .
A numeric value	The parameter can bind to a number type by using an integer value. For example: <code>method="goldCustomer(true, 123)"</code> .
A null value	The parameter can bind as a <code>null</code> value. For example: <code>method="goldCustomer(true, 123, null)"</code> .
A literal	The parameter can bind to a string type by using a literal value. For

value	<code>example: method="goldCustomer(true, 123, 'James Strachan')".</code>
A simple expression	The parameter can bind to any type as a Simple expression. For example: <code>method="goldCustomer(true, \${header.customerId}, \${body.customer.name})".</code>

A value that can't apply to any of the preceding rules would cause Camel to throw an exception at runtime, stating a parameter-binding error.

Bean-binding summary

Camel's rules for bean parameter binding can be summarized as follows:

- All parameters having a Camel annotation will be bound (tables [4.3](#) and [4.4](#)).
- All parameters of a Camel built-in type will be bound (table [4.2](#)).
- The first parameter is assumed to be the message in body (if not already bound).
- If a method name was given containing parameter-binding details, those will be used (table [4.5](#)).
- All remaining parameters will be unbound, and Camel will pass in empty values.

You've seen all there is to bean binding. Camel has a flexible mechanism that adapts to your existing beans, and when you have multiple parameters, Camel provides annotations to bind the parameters properly.

Camel makes it easy to call Java beans from your routes, allowing you to invoke your business logic, or as you saw in chapter 3, to perform message translation using Java code. On

top of that, Camel also makes it easy to use beans as decision makers during routing.

4.6 Using beans as predicates and expressions

Some enterprise integration patterns (EIPs), such as the Content-Based Router and Message Filter, use predicates to determine how they should process messages. Other EIPs, such as the Recipient List, Dynamic Router, and Idempotent Consumer, require using expressions. This section covers how to use Java beans as predicates or expressions with those EIPs.

First, let's briefly review Camel predicates and expressions. A *predicate* is an expression that evaluates as a Boolean; its return value is either `true` or `false`, as depicted in the Camel predicate API:

```
boolean matches(Exchange exchange);
```

An expression, on the other hand, evaluates to anything; its return value is a `java.lang.Object`, as the following Camel API defines:

```
Object evaluate(Exchange exchange);
```

4.6.1 USING BEANS AS PREDICATES IN ROUTES

As you learned in chapter 2, one of the most commonly used EIP patterns is the content-based router. This pattern uses one or more predicates to determine how messages are routed. If a predicate matches, the message is routed down the given path.

The example we'll use is a customer order system that routes orders from gold, silver, and regular customers (`0–999` = gold, `1000–4999` = silver, `5000+` = regular). A bean is used to determine which kind of customer level the routed message is from. A simple implementation is shown in the following listing.

Listing 4.10 Using a bean with methods as a predicate in Camel

```
public class CustomerService {  
    public boolean isGold(@JsonPath("$.order.loyaltyCode")  
int id) { ①
```

①

Method to determine whether it's a gold customer

```
        return id < 1000;  
    }
```

```
    public boolean isSilver(@JsonPath("$.order.loyaltyCode")  
int id) { ②
```

②

Method to determine whether it's a silver customer

```
        return id >= 1000 && id < 5000;  
    }  
}
```

The bean implements two methods, `isGold` ① and `isSilver` ②, both returning a Boolean that allows Camel to use the methods as predicates. Each method uses bean parameter binding to map from the Camel message to the customer ID. This example continues from the previous JSON example and uses the `@JsonPath` annotation to extract the customer ID from the message body in JSON format.

TIP The bean in listing 4.10 uses the bean parameter binding covered in section 4.5. Therefore, you can use what you've learned, and instead of `@JsonPath`, the bean could have multiple parameters.

To use the bean as a predicate in the Content-Based Router pattern in Camel is easy, as shown in the following listing.

Listing 4.11 The content-based router uses the bean as a predicate in XML

```
<bean id="customerService"  
      class="camelinaction.CustomerService"/> 1
```

1

Declares the bean so you can refer to the bean in the upcoming route

```
<camelContext id="camel"  
      xmlns="http://camel.apache.org/schema/spring">  
    <route>  
      <from uri="file://target/order"/>  
      <choice>  
        <when>  
          <method ref="customerService"  
                method="isGold"/> 2
```

2

The method call predicate that invokes the isGold method on the bean

```
        <to uri="mock:queue:gold"/>  
      </when>  
      <when>  
        <method ref="customerService"  
              method="isSilver"/> 3
```

3

The method call predicate that invokes the isSilver method on the bean

```
      <to uri="mock:queue:silver"/>  
    </when>  
    <otherwise>  
      <to uri="mock:queue:regular"/>  
    </otherwise>  
  </choice>  
</route>
```

```
</camelContext>
```

As you can see, using a bean as a predicate is easy, by using `<method>` ❷ ❸ to set up the bean as the predicate. Prior to that, the bean needs to be declared using a `<bean>` element 1.

The following listing shows how to implement the same route as in [listing 4.11](#) but using Java DSL.

Listing 4.12 The content-based router using the bean as a predicate in Java DSL

```
public void configure() throws Exception {  
    from("file://target/order")  
        .choice()  
            .when(method(CustomerService.class, "isGold")) ❶
```

❶

The method call predicate that invokes the `isGold` method on the bean

```
                .to("mock:queue:gold")  
                .when(method(CustomerService.class,  
"isSilver")) ❷
```

❷

The method call predicate that invokes the `isSilver` method on the bean

```
                .to("mock:queue:silver")  
                .otherwise()  
                    .to("mock:queue:regular");  
}
```

The route in [listing 4.12](#) is almost identical to [listing 4.11](#). But this time we show a slight variation that allows you to refer to the bean by its class name, `CustomerService.class` ❶ ❷. When doing this, Camel may be required to create a new instance of the bean while the route is being created, and therefore the bean is required to have a default no-argument constructor. But if the method `isGold` or `isSilver` is a static method, Camel won't create a new bean instance, but will invoke the static methods directly.

The Java DSL could also, as in [listing 4.11](#), refer to the bean by an ID, which would require using the bean name instead of the class name, as shown in this snippet:

```
.when(method("customerService", "isGold"))
    .to("mock:queue:gold")
```

In Java DSL, there's more, as you can combine multiple predicates into compound predicates.

USING COMPOUND PREDICATES IN JAVA

This is an exclusive feature in Java only that allows you to combine one or more predicates into a compound predicate. This can be used to logically combine (and, or, not) multiple predicates, even if they use different languages—for example, combining XPath, Simple, and Bean, as shown in [listing 4.13](#).

Listing 4.13 Using PredicateBuilder to build a compound predicate using XPath, Simple, and method call together

```
return new RouteBuilder() {
    public void configure() throws Exception {
        Predicate valid = PredicateBuilder.and( ❶
```

❶

Uses PredicateBuilder to combine the predicates using and

```
        xpath("/book/title = 'Camel in Action'), ❷
```

❷

The XPath predicate that tests whether the book title is Camel in Action

```
        simple("${header.source} == 'batch'"), ❸
```

❸

The Simple predicate that tests whether header.source has the value batch

```
        not(method(CompoundPredicateTest.class,
```

```
"isAuthor")));
```

④

④

The method call predicate calls the isAuthor method that's negated using not, and therefore should return false

```
from("direct:start")
    .validate(valid)
```

⑤

Uses the compound predicate in the Camel route using the Validate EIP pattern

```
        .to("mock:valid");
    }
};
```

The substance is the

org.apache.camel.builder.PredicateBuilder ① that has a number of builder methods to combine predicates. We use and, which means all three ② ③ ④ predicates must return true for the compound predicate to return true. If one of them returns false, the compound predicate response is also false.

The source code for the book includes this example in the chapter4/predicate directory, which can be executed using the following Maven goal:

```
mvn test -Dtest=CompoundPredicateTest
```

Beans can also be used as expressions in routes, which is the next topic.

4.6.2 USING BEANS AS EXPRESSIONS IN ROUTES

This section covers a common use case with Camel: using a *dynamic to* route. You'll see how to route a message in Camel to a destination that's dynamically computed at runtime with information from the message itself.

In the *Enterprise Integration Patterns* book, the Recipient List pattern best describes the *dynamic to*, illustrated in [figure 4.10](#).

The Recipient List EIP pattern in Camel is a versatile and flexible implementation that offers many features and functions. This EIP is covered in more detail in the following chapter.

To use a bean with the Recipient List pattern, you use a simple use case with a customer service system that routes orders depending on geographical region of the customer. A naive bean could be implemented in a few lines of code, as shown here:

```
public class CustomerService {  
  
    public String region(@JsonPath("$.order.customerId") int customerId) {  
        if (customerId < 1000) {  
            return "US";  
        } else if (customerId < 2000) {  
            return "EMEA";  
        } else {  
            return "OTHER";  
        }  
    }  
}
```

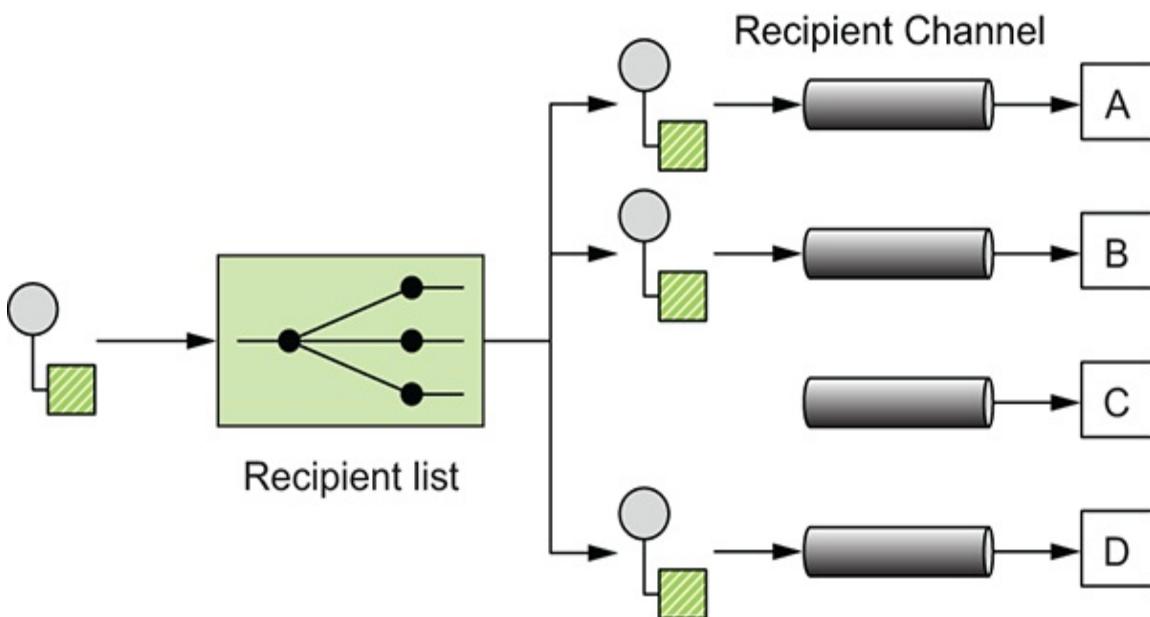


Figure 4.10 Recipient List EIP sends a copy of the same message to numerous dynamic computed destinations. The *dynamic to* is just a single dynamic

computed destination.

To use the bean as an expression in the Recipient List pattern in Camel is also easy, as shown in the following listing.

Listing 4.14 Using a bean as an expression during routing with the recipient list to act as a dynamic to

```
<bean id="customerService"  
      class="camelaction.CustomerService"/>
```

①

①

Bean to be used as expression implementing logic to determine customer geographical location

```
<camelContext id="camel"  
      xmlns="http://camel.apache.org/schema/spring">  
    <route>  
      <from uri="file://target/order"/>  
      <setHeader headerName="region">  
        <method ref="customerService" method="region"/>
```

②

②

Calling the bean to set a header with the region of the customer

```
</setHeader>  
  <recipientList>  
    <simple>mock:queue:${header.region}</simple>
```

③

③

Dynamic to using recipient list to route to a queue with the name of the region

```
  </recipientList>  
  </route>  
</camelContext>
```

First, you need to set up the bean as a `<bean>` ① so you can refer to the bean by using its ID, `customerService`. Then the bean is invoked to return the geographical region from which the

customer order is placed ②. The bean will return either US, EMEA, or OTHER, depending on the region. Then the message is routed using the recipient list ③ to a single destination, hence it's also referred as *dynamic to*. If the customer is from US, the destination would be `mock:queue:US`, and for EMEA it would be `mock:queue:EMEA`.

Listing 4.14 could be implemented in Java DSL with fewer lines of code, as shown here:

```
from("file://target/order")
    .setHeader("region", method(CustomerService.class,
"region"))
    .recipientList(simple("mock:queue:${header.region}"));
```

This book's source code includes this example in the chapter4/expression directory, which you can run using the following Maven goals:

```
mvn test -Dtest=JsonExpressionTest
mvn test -Dtest=SpringJsonExpressionTest
```

In the example, you didn't call the bean from the recipient list, but instead computed the region as a header. You could omit this and call the bean directly from the recipient list, as the following code shows:

```
from("file://target/order")
    .recipientList(simple("mock:queue:" +
"${bean:camelInAction.CustomerService?
method=region}"));
```

And using XML DSL:

```
<from uri="file://target/order"/>
<recipientList>
    <simple>
        mock:queue:${bean:camelInAction.CustomerService?
method=region}
    </simple>
</recipientList>
```

A simpler dynamic to

The Recipient List is the EIP pattern that allows you to send messages to one or more dynamic endpoints, and it has been our answer in Camel since Camel was created. But over the years, we've learned that some Camel users weren't familiar with the Recipient List pattern, and therefore couldn't find a way to easily send a message to a single Camel endpoint that was dynamically computed with information from the message. As a new user to Camel, you quickly get the hang of using `<from>` and `<to>` in your routes, which are key patterns. To make this easier in Camel, we introduced `<toD>` as an alternative to `<recipientList>`.

USING `toD` AS DYNAMIC TO

If you need to send a message to a single dynamic computed endpoint, you should favor using `toD`. It's specially designed for the single-destination use case, whereas Recipient List is a much more elaborate EIP pattern that in those use cases can be overkill. `toD` has the following characteristics:

- Can send to only one dynamic destination
- Can use only the Simple language as the expression to compute the dynamic endpoint

Dynamic `to` has specifically been designed to work like `to`, but can send the message to a dynamic computed endpoint by using the Simple language. If you have any other needs, use the more powerful Recipient List EIP.

The previous example in [listing 4.14](#) can be simplified to use `<toD>`, as shown in the following listing.

[Listing 4.15](#) Using a bean as an expression during routing

with dynamic to

```
<bean id="customerService"
      class="camelaction.CustomerService"/>
```

Bean to be used as expression, implementing logic to determine customer geographical location

```
<camelContext id="camel"
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://target/order"/>
    <setHeader headerName="region">
      <method ref="customerService" method="region"/>
```

Calling the bean to set a header with the region of the customer

```
    </setHeader>
    <toD uri="mock:queue:${header.region}"/>
```

Dynamic to route to a queue with the name of the region

```
  </route>
</camelContext>
```

The example is only a mere three lines of code when using Java DSL, as shown here:

```
from("file://target/order")
  .setHeader("region", method(CustomerService.class,
"region"))
  .toD("mock:queue:${header.region}");
```

And if you want to take the example down to two lines of code, you can call the bean directly instead of setting the header:

```
from("file://target/order")
  .toD("mock:queue:${bean:customerService?method=region}");
```

This book's source code includes this example in the chapter4/expression directory, and you can run the example by

using the following Maven goals:

```
mvn test -Dtest=JsonToDExpressionTest  
mvn test -Dtest=SpringJsonToDExpressionTest
```

That's all we have to say about using beans as predicates and expressions. In fact, we've reached the end of this chapter.

4.7 Summary and best practices

We've now covered another cornerstone of using beans with Camel. It's important that end users of Camel can use the POJO programming model and have Camel easily use those beans (POJOs). Beans are just Java code, which is a language you're likely to feel comfortable using. If you hit a problem that you can't work around or figure out how to resolve using Camel and EIPs, you can always resort to using a bean and letting Camel invoke it.

We unlocked the algorithm used by Camel to select which method to invoke on a bean. You learned why this is needed: Camel must resolve method selection at runtime, whereas regular Java code can link method invocations at compile time.

We also covered what bean parameter binding is and how to bind a Camel exchange to any bean method and its parameters. You learned how to use annotations to provide fine-grained control over the bindings, and even how Camel can help bind XPath or JsonPath expressions to parameters, which is a great feature when working with XML or JSON messages.

Let's pull out some of the key practices you should take away from this chapter:

- *Use beans*—Beans are Java code, and they give you all the horsepower of Java.
- *Use loose coupling*—Prefer using beans that don't have a strong dependency on the Camel API. Camel is capable of adapting to existing bean-method signatures, so you can use any existing API you may have, even if it has no dependency on the Camel API.

Unit testing is also easier because your beans don't depend on any Camel API. You can even have developers with no Camel experience develop the beans, and then have developers with Camel experience use those beans.

- *Use method facades*—If calling your existing beans in a loosely coupled fashion seems too difficult, or you have to specify too many bean parameter mappings or want to avoid introducing Camel annotations on your existing bean, you can use method facades. You can create a new bean as a facade, and use Java code to implement the mapping between Camel and your existing bean.
- *Prefer simple method signatures*—Camel bean binding is much simpler when method signatures have as few parameters as possible.
- *Specify method names*—Tell Camel which method you intend to invoke, so Camel doesn't have to figure it out. You can also use `@Handler` in the bean to tell Camel which method it should pick and use.
- *Favor parameter binding using method-signature syntax*—When calling methods on POJOs from Camel routes, it's often easier to specify the parameter binding in the method-name signature in the route that closely resembles Java code. This makes it easier for other users of Camel to understand the code.
- *Use the powers of Java as predicates or expressions*—When you need to define predicates or expressions when using a more powerful language, consider using plain-old Java code to implement this logic. The Java code can be loosely coupled from Camel and allows for easier unit testing the code, isolated from Camel.

We've now covered three crucial features of integration kits: routing, transformations, and using beans. In chapter 2, you were exposed to some of Camel's routing capabilities by using standard EIPs. In the next chapter, you'll look at some of the more complex EIPs available in Camel.

5

Enterprise integration patterns

This chapter covers

- The Aggregator EIP
- The Splitter EIP
- The Routing Slip EIP
- The Dynamic Router EIP
- The Load Balancer EIP

Today's businesses aren't run on a single monolithic system, and most businesses have a full range of disparate systems. There's an ever-increasing demand for those systems to integrate with each other and with external business partners and government systems.

Let's face it: integration is hard. To help deal with its complexity, enterprise integration patterns (EIPs) have become the standard way to describe, document, and implement complex integration problems. We explain the patterns we discuss in this book, but to learn more about them and others, see the Enterprise Integration Patterns website and the associated book: www.enterpriseintegrationpatterns.com.

5.1 Introducing enterprise integration

patterns

Apache Camel implements EIPs, and because the EIPs are essential building blocks in the Camel routes, you'll bump into EIPs throughout this book, starting in chapter 2. It would be impossible for this book to cover all the EIPs Camel supports, which currently total about 70 patterns. This chapter is devoted to covering five of the most powerful and feature-rich patterns, listed in table 5.1.

Table 5.1 EIPs covered in this chapter

Pattern	Summary
Aggregator	Used to combine results of individual but related messages into a single outgoing message. You can view this as the <i>reverse</i> of the Splitter pattern. This pattern is covered in section 5.2.
Splitter	Used to split a message into pieces that are routed separately. This pattern is covered in section 5.3.
Routing Slip	Used to route a message in a series of steps; the sequence of steps isn't known at design time and may vary for each message. This pattern is covered in section 5.4.
Dynamic Router	Used to route messages with a dynamic router dictating where the message goes. This pattern is covered in section 5.5.
Load Balancer	Used to balance the load to a given endpoint by using a variety of balancing policies. This pattern is covered in section 5.6.

Let's look at these patterns in more detail.

5.1.1 THE AGGREGATOR AND SPLITTER EIPS

The first two patterns listed in table 5.1 are related. The Splitter can split a single message into multiple submessages, and the Aggregator can combine those submessages back into a single message. They're opposite patterns.

The EIPs allow you to build patterns *LEGO style*, which means that patterns can be combined together to form new patterns. For example, you can combine the Splitter and the Aggregator into what is known as the Composed Message Processor EIP, as illustrated in figure 5.1.

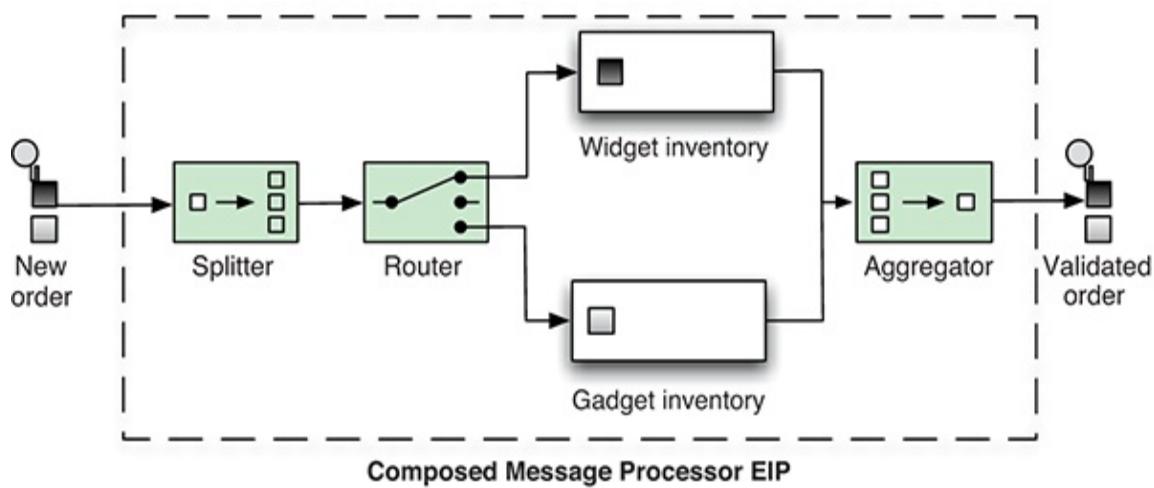


Figure 5.1 The Composed Message Processor EIP splits up the message, routes the submessages to the appropriate destinations, and reaggregates the response back into a single message.

The Aggregator EIP is likely the most sophisticated and most advanced EIP implemented in Camel. It has many use cases, such as aggregating incoming bids for auctions or throttling stock quotes.

5.1.2 THE ROUTING SLIP AND DYNAMIC ROUTER EIPS

A question that's often asked on the Camel user mailing list is how to route messages dynamically. The answer is to use EIPs such as Recipient List, Routing Slip, and Dynamic Router. Chapter 2 covered Recipient List, and this chapter will show you how to use the Routing Slip and Dynamic Router patterns.

5.1.3 THE LOAD BALANCER EIP

The EIP book doesn't list the Load Balancer, which is a pattern implemented in Camel. Suppose you route PDF messages to network printers, and those printers come and go online. You

can use the Load Balancer to send the PDF messages to another printer if one printer is unresponsive.

That covers the five EIPs covered in this chapter. It's now time to look at the first one in detail, the Aggregator EIP.

5.2 The Aggregator EIP

The Aggregator EIP is important and complex, so we'll cover it thoroughly. The Aggregator combines many related incoming messages into a single aggregated message, as illustrated in figure 5.2.



Figure 5.2 The Aggregator stores incoming messages until it receives a complete set of related messages. Then the Aggregator publishes a single message distilled from the individual messages.

The Aggregator receives a stream of messages and identifies messages that are related, which are then aggregated into a single combined message. After a completion condition occurs, the aggregated message is sent to the output channel for further processing. The next section covers how this process works in detail.

Example uses of Aggregator

The Aggregator EIP supports many use cases, such as the loan broker example from the EIP book, in which brokers send loan requests to multiple banks and aggregate the replies to determine the *best deal*.

You could also use the Aggregator in an auction system to aggregate current bids. Also imagine a stock market system that continuously receives a stream of stock

quotes, and you want to throttle this to publish the latest quote every five seconds. This can be done by using the Aggregator to choose the latest message and thus trigger a completion every five seconds.

When using the Aggregator, you have to pay attention to the following three settings, which must be configured. Failure to do so will cause Camel to fail on startup and report an error regarding the missing configuration:

- *Correlation identifier*—An Expression that determines which incoming messages belong together
- *Completion condition*—A Predicate or time-based condition that determines when the result message should be sent
- *Aggregation strategy*—An AggregationStrategy that specifies how to combine the messages into a single message

In this section, you'll look at a simple example that will aggregate messages containing alphabetic characters, such as *A*, *B*, and *C*. This will keep things simple, making it easier to follow what's going on. The Aggregator is equally equipped to work with big loads, but that can wait until we've covered the basic principles.

5.2.1 USING THE AGGREGATOR EIP

Suppose you want to collect any three messages and combine them. Given three messages containing *A*, *B*, and *C*, you want the aggregator to output a single message containing *ABC*.

Figure 5.3 shows how this works. When the first message with correlation identifier 1 arrives, the aggregator initializes a new aggregate and stores the message inside the aggregate. In this example, the completion condition is the aggregation of three messages, so the aggregate isn't yet complete. When the second message with correlation identifier 1 arrives, the EIP adds it to the already existing aggregate. The third message specifies a different correlation identifier value of 2, so the aggregator starts a new aggregate for that value. The fourth message relates to the

first aggregate (identifier 1), so the aggregate has now aggregated three messages, and the completion condition is fulfilled. As a result, the aggregator marks the aggregate as complete and publishes the resulting message.

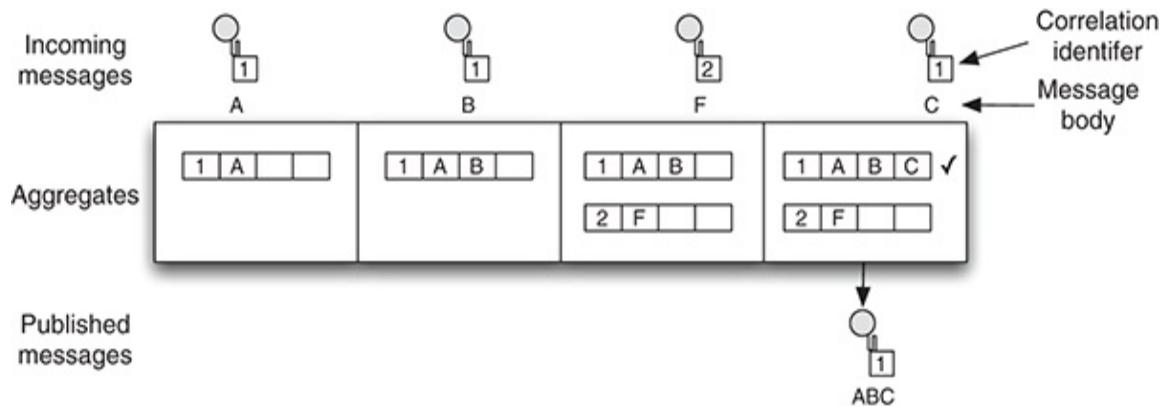


Figure 5.3 The Aggregator EIP in action, with partial aggregated messages updated with arriving messages

As mentioned before, three configurations are in play when using the Aggregator EIP: correlation identifier, completion condition, and aggregation strategy. To understand how these three are specified and how they work, let's start with the example of a Camel route in the Java DSL (with the configurations in bold):

```
public void configure() throws Exception {
    from("direct:start")
        .log("Sending ${body} with correlation key
${header.myId}")
        .aggregate(header("myId"), new
MyAggregationStrategy())
            .completionSize(3)
            .log("Sending out ${body}")
            .to("mock:result");
```

The correlation identifier is `header("myId")`, and it's a Camel Expression. It returns the header with the key `myId`. The second configuration element is the `AggregationStrategy`, which is a class. We'll cover this class in more detail in a moment. Finally, the completion condition is based on size (there are seven kinds of completion conditions, listed in table 5.3). It states that when three messages have been aggregated, the completion should

trigger.

The same example in XML is as follows:

```
<bean id="myAggregationStrategy"
      class="camelaction.MyAggregationStrategy"/>

<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <log message="Sending ${body} with key
${header.myId}"/>
        <aggregate strategyRef="myAggregationStrategy"
completionSize="3">
            <correlationExpression>
                <header>myId</header>
            </correlationExpression>
            <log message="Sending out ${body}"/>
            <to uri="mock:result"/>
        </aggregate>
    </route>
</camelContext>
```

The XML snippet is a little different from the Java DSL because you define AggregationStrategy by using the `strategyRef` attribute on the `<aggregate>` tag. This refers to a `<bean>`, which is listed in the top of the XML file. The completion condition is also defined as a `completionSize` attribute. The most noticeable difference is the way the correlation identifier is defined. In XML, it's defined using the `<correlationExpression>` tag, which has a child tag that includes the Expression.

The book's source code contains this example in the `chapter5/aggregator` directory. You can run the examples by using the following Maven goals:

```
mvn test -Dtest=AggregateABCTest
mvn test -Dtest=SpringAggregateABCTest
```

The examples use the following unit-test method:

```
public void testABC() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedBodiesReceived("ABC");
```

```

        template.sendBodyAndHeader("direct:start", "A", "myId",
1);
        template.sendBodyAndHeader("direct:start", "B", "myId",
1);
        template.sendBodyAndHeader("direct:start", "F", "myId",
2);
        template.sendBodyAndHeader("direct:start", "C", "myId",
1);
        assertMockEndpointsSatisfied();
}

```

This unit test sends the same messages as shown in [figure 5.3](#)—four messages in total. When you run the test, you’ll see the output on the console:

```

INFO route1 - Sending A with correlation key 1
INFO route1 - Sending B with correlation key 1
INFO route1 - Sending F with correlation key 2
INFO route1 - Sending C with correlation key 1
INFO route1 - Sending out ABC

```

Notice that the console output matches the sequence in which the messages were aggregated in the example from [figure 5.3](#). As you can see from the console output, the messages with correlation key 1 were completed, because they met the completion condition, which was size based on three messages. The last line of the output shows the published message, which contains the letters *ABC*.

So what happens with the *F* message? Well, its completion condition hasn’t been met, so it waits in the aggregator. You could modify the test method to send an additional two messages to complete that second group as well:

```

template.sendBodyAndHeader("direct:start", "G", "myId", 2);
template.sendBodyAndHeader("direct:start", "H", "myId", 2);

```

Let’s now turn our focus to how the Aggregator EIP combines the messages, which causes the *A*, *B*, and *C* messages to be published as a single message. This is where the `AggregationStrategy` comes into the picture, because it orchestrates this.

USING AGGREGATIONSTRATEGY

The `AggregationStrategy` class is located in the `org.apache.camel.processor.aggregate` package, and it defines a single method:

```
public interface AggregationStrategy {  
    Exchange aggregate(Exchange oldExchange, Exchange  
newExchange);  
}
```

If you're having a déjà vu moment, it's most likely because `AggregationStrategy` is also used by the Content Enricher EIP, which we covered in chapter 3.

The following listing shows the strategy used in the previous example.

Listing 5.1 `AggregationStrategy` for merging messages

```
public class MyAggregationStrategy implements  
AggregationStrategy {  
    public Exchange aggregate(Exchange oldExchange,  
Exchange newExchange) {  
        if (oldExchange == null) {  
            return newExchange; 1  
        }
```

1

Occurs for a new group

```
}  
String oldBody = oldExchange.getIn() 2
```

2

Combines message bodies

```
.getBody(String.class); 2  
String newBody = newExchange.getIn() 2  
.getBody(String.class); 2  
String body = oldBody + newBody; 2  
oldExchange.getIn().setBody(body); 1
```

3

Replaces message body with combined message bodies

```
        return oldExchange;
    }
}
```

At runtime, the aggregate method is invoked every time a new message arrives. In this example, it'll be invoked four times: one for each arriving message *A*, *B*, *F*, and *C*. To show how this works, table 5.2 lists the invocations as they'd happen.

Table 5.2 Sequence of invocations of aggregate method occurring at runtime

Arrived	oldExchange	newExchange	Description
A	null	A	The first message arrives for the first group.
B	A	B	The second message arrives for the first group.
F	null	F	The first message arrives for the second group.
C	AB	C	The third message arrives for the first group.

Notice in table 5.2 that the `oldExchange` parameter is `null` on two occasions. This occurs when a new correlation group is formed (no preexisting messages have arrived with the same correlation identifier). In this situation, you want to return the message as is, because there are no other messages to combine it with ❶.

On the subsequent aggregations, neither parameter is `null`, so you need to merge the data into one Exchange. In this example, you grab the message bodies and add them together ❷. Then you replace the existing body in `oldExchange` with the updated body ❸.

NOTE The Aggregator EIP uses synchronization, which ensures that `AggregationStrategy` is thread safe—only one thread is invoking the aggregate method at any time. The

Aggregator also ensures ordering, which means the messages are aggregated in the same order as they're sent into the Aggregator.

You should now understand the principles of how the Aggregator works. For a message to be published from the Aggregator, a completion condition must have been met. In the next section, we discuss this and review the conditions Camel provides out of the box.

5.2.2 COMPLETION CONDITIONS FOR THE AGGREGATOR

Completion conditions play a bigger role in the Aggregator than you might think. Imagine a situation in which a condition never occurs, causing aggregated messages never to be published. For example, suppose the C message never arrived in the example in section 5.2.1. To remedy this, you could add a time-out condition that reacts if all messages aren't received within a certain time period.

To cater for that situation and others, Camel provides seven completion conditions, listed in table 5.3. You can mix and match them according to your needs.

Table 5.3 Completion conditions provided by the Aggregator EIP

Condition	Description
completionSize	Defines a completion condition based on the number of messages aggregated together. You can either use a fixed value (<code>int</code>) or use an Expression to dynamically decide a size at runtime.
completionTimeout	Defines a completion condition based on an inactivity time-out. This condition triggers if a correlation group has been inactive longer than the specified period. Time-outs are scheduled for each correlation group, so the time-out is individual to each group. You can either use a fixed value (<code>long</code>) or an Expression to dynamically decide a time-out at runtime. The period is defined in milliseconds. You can't use this condition together with the <code>completionInterval</code> .

completionInInterval	<p>Defines a completion condition based on a scheduled interval. This condition triggers periodically. There's a single scheduled time-out for <i>all</i> correlation groups, which causes all groups to complete at the same time.</p> <p>The period (<code>long</code>) is defined in milliseconds. You can't use this condition together with the <code>completionTimeout</code>.</p>
completionOnPredicate	<p>Defines a completion condition based on whether the <code>Predicate</code> matched. See also the <code>eagerCheckCompletion</code> option in table 5.5.</p> <p>This condition is enabled automatically if <code>aggregationStrategy</code> implements either <code>Predicate</code> or <code>PreCompletionAwareAggregationStrategy</code>. In the case of <code>PreCompletionAwareAggregationStrategy</code>, this gives you the ability to complete the aggregation group on receipt of a new <code>Exchange</code> and start a new group with the new <code>Exchange</code>.</p>
completionFromBatchConsumer	<p>Defines a completion condition that's applicable only when the arriving <code>Exchanges</code> are coming from a <code>BatchConsumer</code> (http://camel.apache.org/batch-consumer.html). Numerous components support this condition, such as Atom, File, FTP, HBase, Mail, MyBatis, JClouds, SNMP, SQL, SQS, S3, and JPA.</p>
forceCompletionOnStop	<p>Defines a completion condition that will complete all correlation groups on shutdown of <code>CamelContext</code>.</p>
aggregateController	<p>By using <code>AggregateController</code>, you can control group completion externally via Java calls into the controller or via JMX.</p>

The Aggregator supports using multiple completion conditions, such as using both the `completionSize` and `completionTimeout` conditions. When using multiple conditions, though, the winner takes all: the completion condition that completes first will result in the message being published.

NOTE The book's source code contains examples in the `chapter5/aggregator` directory for all conditions; you can refer to them for further details. Also the Aggregator documentation on the Camel website has more details:

<http://camel.apache.org/aggregator2>.

We'll now look at how to use multiple completion conditions.

USING MULTIPLE COMPLETION CONDITIONS

The book's source code contains an example in the chapter5/aggregator directory showing how to use multiple completion conditions. You can run the example by using the following Maven goals:

```
mvn test -Dtest=AggregateXMLTest  
mvn test -Dtest=SpringAggregateXMLTest
```

The route in the Java DSL is as follows:

```
import static  
org.apache.camel.builder.xml.XPathBuilder.xpath;  
  
public void configure() throws Exception {  
    from("direct:start")  
        .log("Sending ${body}")  
        .aggregate(xpath("/order/@customer"), new  
MyAggregationStrategy())  
            .completionSize(2).completionTimeout(5000)  
            .log("Sending out ${body}")  
            .to("mock:result");  
}
```

As you can see from the bold code in the route, using a second condition is just a matter of adding a completion condition.

The same example in XML is shown here:

```
<bean id="myAggregationStrategy"  
      class="camelaction.MyAggregationStrategy"/>  
  
<camelContext  
  xmlns="http://camel.apache.org/schema/spring">  
  <route>  
    <from uri="direct:start"/>  
    <log message="Sending ${body}" />  
    <aggregate strategyRef="myAggregationStrategy"  
              completionSize="2"
```

```

completionTimeout="5000">
    <correlationExpression>
        <xpath>/order/@customer</xpath>
    </correlationExpression>
    <log message="Sending out ${body}"/>
    <to uri="mock:result"/>
</aggregate>
</route>
</camelContext>

```

If you run this example, it'll use the following test method:

```

public void testXML() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(2);
    template.sendBody("direct:start",
        "<order name=\"motor\" amount=\"1000\""
    customer="\"honda\"/>");
    template.sendBody("direct:start",
        "<order name=\"motor\" amount=\"500\""
    customer="\"toyota\"/>");
    template.sendBody("direct:start",
        "<order name=\"gearbox\" amount=\"200\""
    customer="\"toyota\"/>");
    assertMockEndpointsSatisfied();
}

```

This example should cause the aggregator to publish two outgoing messages, as shown in the following console output—one for Honda and one for Toyota:

```

09:37:35 - Sending <order name="motor" amount="1000"
customer="honda"/>
09:37:35 - Sending <order name="motor" amount="500"
customer="toyota"/>
09:37:35 - Sending <order name="gearbox" amount="200"
customer="toyota"/>
09:37:35 - Sending out
    <order name="motor" amount="500"
customer="toyota"/>
    <order name="gearbox" amount="200"
customer="toyota"/>
09:37:41 - Sending out
    <order name="motor" amount="1000"
customer="honda"/>

```

If you look closely at the test method and the output from the

console, you should notice that the Honda order arrived first, but it was the last to be published. This is because its completion was triggered by the time-out, which was set to 5 seconds. In the meantime, the Toyota order had its completion triggered by the size of two messages, so it was published first.

TIP The Aggregator EIP allows you to use as many completion conditions as you like. But the `completionTimeout` and `completionInterval` conditions can't be used at the same time.

Using multiple completion conditions makes good sense if you want to ensure that aggregated messages eventually get published. For example, the time-out condition ensures that after a period of inactivity, the message will be published. In that regard, you can use the time-out condition as a fallback condition, with the price being that the published message will be only partly aggregated. Suppose you expect two messages to be aggregated into one, but you receive only one message; the next section reveals how you can tell which condition triggered the completion.

AGGREGATED EXCHANGE PROPERTIES

Camel enriches the published Exchange with the completion details listed in table 5.4.

Table 5.4 Properties on the Exchange related to aggregation

Property	Type	Description
<code>Exchange.AGGREGATED_SIZE</code>	<code>int</code>	The total number of arrived messages aggregated.
<code>Exchange.AGGREGATED_COMPLETED_BY</code>	<code>string</code>	The condition that triggered the completion. Possible values are <code>size</code> , <code>timeout</code> , <code>interval</code> , <code>predicate</code> , <code>force</code> , <code>strategy</code> , and <code>consumer</code> .

	n g	The consumer value represents the completion from batch consumer.
Exchange.AGGREGATED_CORRELATION_KEY	s t r i n g	The correlation identifier as a String.
Exchange.AGGREGATED_TIMEOUT	l o n g	The time-out in milliseconds as set by the completion time-out.
Exchange.AGGREGATION_COMPLETE_CURRENT_GROUP	b o o l e a n	Set this to true to complete the current group.
Exchange.AGGREGATION_COMPLETE_ALL_GROUPS	b o o l e a n	Set this to true to complete all groups and ignore the current Exchange.
Exchange.AGGREGATION_COMPLETE_ALL_GROUPS_INCLUSIVE	b o o l e a n	Set this to true to complete all groups and include the current Exchange.

The information listed in table 5.4 allows you to know how a published aggregated Exchange was completed, and how many messages were combined. For example, you could log which condition triggered the completion by adding this to the Camel route:

```
.log("Completed by ${property.CamelAggregatedCompletedBy}")
```

This information might come in handy in your business logic, when you need to know whether all messages were aggregated.

You can tell this by checking the `AGGREGATED_COMPLETED_BY` property, which could contain several values, including `size` and `timeout`. If the value is `size`, all the messages were aggregated; if the value is `timeout`, a time-out occurred, and not all expected messages were aggregated. The Aggregator has additional configuration options that you may need to use. For example, you can specify how it should react when an arrived message contains an invalid correlation identifier.

ADDITIONAL CONFIGURATION OPTIONS

The Aggregator is the most sophisticated EIP implemented in Camel, and table 5.5 lists the additional configuration options you can use to tweak it to fit your needs.

Table 5.5 Additional configuration options available for the Aggregator EIP

Configuration option	Default	Description
<code>eagerCheckCompletion</code>	<code>false</code>	<p>This option specifies whether to eager-check for completion. Eager-checking means Camel will check for completion conditions before aggregating. By default, Camel checks for completion after aggregation.</p> <p>This option is used to control how the <code>completionPredicate</code> condition behaves. If the option is <code>false</code>, the completion predicate will use the aggregated Exchange for evaluation. If <code>true</code>, the incoming Exchange will be used for evaluation.</p>
<code>closeCorrelationOnKeyCompletion</code>		<p>This option determines whether a given correlation group should be marked as closed when it's completed. If a correlation group is closed, any subsequent arriving Exchanges are rejected and a <code>ClosedCorrelationKeyException</code> is thrown. You need to use an <code>Integer</code> parameter that represents a maximum bound for a least recently used (LRU) cache. This keeps track of closed correlation keys. Note that this cache is in-memory only and will be reset if Camel is restarted.</p>
<code>ignoreInvalidCorrelationKeys</code>	<code>false</code>	<p>This option specifies whether to ignore invalid correlation keys. By default, Camel throws a <code>CamelExchangeException</code> for invalid keys. You can suppress this by setting this option to <code>true</code>, in which case</p>

	e	Camel skips the invalid message.
timeout Checker ExecutorService / timeout Checker ExecutorServiceRef		Specifies a ScheduledExecutorService to be used as the thread pool when checking for timeout-based completion conditions.
optimisticLocking	false	Turns on optimistic locking of the aggregation repository. The aggregation repository must implement org.apache.camel.spi.OptimisticLockingAggregationRepository.
optimisticLockRetryPolicy		Specifies OptimisticLockRetryPolicy used to control how lock retries occur.

If you want to learn more about the configuration options listed in table 5.5, there are examples for most in the book's source code in the chapter5/aggregator directory. You can run test examples by using the following Maven goals:

```
mvn test -Dtest=AggregateABCEagerTest
mvn test -Dtest=SpringAggregateABCEagerTest
mvn test -Dtest=AggregateABCCloseTest
mvn test -Dtest=SpringAggregateABCCloseTest
mvn test -Dtest=AggregateABCInvalidTest
mvn test -Dtest=SpringAggregateABCInvalidTest
mvn test -Dtest=AggregateABCGroupTest
mvn test -Dtest=SpringAggregateABCGroupTest
mvn test -Dtest=AggregateTimeoutThreadpoolTest
mvn test -Dtest=SpringAggregateTimeoutThreadpoolTest
```

Next, we'll look at implementing aggregation strategies without using any Camel API at all.

USING POJOs FOR THE AGGREGATIONSTRATEGY

So far you've seen that you can customize how Exchanges are aggregated by implementing AggregationStrategy in a custom

class. There's a slightly cleaner way of doing this, however, without using Camel APIs at all. As with Camel's bean integration, you can provide the aggregator with a POJO to act as AggregationStrategy. Camel handles injecting one or all of the message body, headers, and Exchange properties. Taking a look at AggregationStrategy used in [listing 5.1](#), you can provide an equivalent POJO version as follows:

```
public class MyAggregationStrategyPojo {  
    public String concat(String oldBody, String newBody) {  
        if (newBody != null) {  
            return oldBody + newBody;  
        } else {  
            return oldBody;  
        }  
    }  
}
```

As you can see, it's much cleaner than AggregationStrategy in [listing 5.1](#). Similarly, though, it has two parameters: one for the existing aggregated message (oldBody) and one for the incoming message (newBody). If you need either the message headers and/or the Exchange properties, you can use method signatures such as these:

```
public String concat(String oldBody, Map oldHeaders,  
                     String newBody, Map newHeaders);  
public String concat(String oldBody, Map oldHeaders, Map  
oldProperties,  
                     String newBody, Map newHeaders,  
Map newProperties);
```

Notice that you have to add more parameters in pairs and that order is important. The first parameter is the body, the second is the header, and third is exchange properties.

The Camel route looks different from before as well. When referencing the preceding AggregationStrategy POJO, your route looks like this:

```
import org.apache.camel.util.toolbox.AggregationStrategies;  
  
public void configure() throws Exception {
```

```

        from("direct:start")
            .log("Sending ${body} with correlation key
${header.myId}")
            .aggregate(header("myId"),
AggregationStrategies.bean(new
MyAggregationStrategyPojo()))
            .completionSize(3)
                .log("Sending out ${body}")
            .to("mock:result");
}

```

As you can see in bold in the preceding code, you use the `AggregationStrategies` utility class to convert the POJO into an `AggregationStrategy`. Many more methods are available in this class as well. For instance, you can pass in a bean reference and select the method you want to use, or even just the class.

The same route in XML is shown here:

```

<bean id="myAggregationStrategy"
      class="camelinaction.MyAggregationStrategy"/>

<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <log message="Sending ${body} with correlation key
${header.myId}"/>
        <aggregate strategyRef="myAggregationStrategy"
completionSize="3">
            <correlationExpression>
                <header>myId</header>
            </correlationExpression>
            <log message="Sending out ${body}"/>
            <to uri="mock:result"/>
        </aggregate>
    </route>
</camelContext>

```

If you want to try this for yourself, examples are provided in the book's source code, in the `chapter5/aggregator` directory. You can run test examples by using the following Maven goals:

```

mvn test -Dtest=AggregatePojoTest
mvn test -Dtest=SpringAggregatePojoTest

```

In the next section, we'll look at solving the problems with persistence. The Aggregator, by default, uses an in-memory repository to hold the current in-progress aggregated messages, and those messages will be lost if the application is stopped or the server crashes. To remedy this, you need to use a persisted repository.

5.2.3 USING PERSISTENCE WITH THE AGGREGATOR

The Aggregator is a stateful EIP because it needs to store the in-progress aggregates until completion conditions occur and the aggregated message can be published. By default, the Aggregator will keep state in memory only. If the application is shut down or the host container crashes, the state will be lost.

To remedy this problem, you need to store the state in a persistent repository. Camel provides a pluggable feature so you can use a repository of your choice. This comes in three flavors:

- `AggregationRepository`—An interface that defines the general operations for working with a repository, such as adding data to and removing data from it. By default, Camel uses `MemoryAggregationRepository`, which is a memory-only repository.
- `RecoverableAggregationRepository`—An interface that defines additional operations supporting recovery. Camel provides several such repositories out of the box, including `JdbcAggregationRepository`, `CassandraAggregationRepository`, `LevelDBAggregationRepository`, and `HazelcastAggregationRepository`. We cover recovery in section 5.2.4.
- `OptimisticLockingAggregationRepository`—An interface that defines additional operations supporting optimistic locking. The `MemoryAggregationRepository` and `JdbcAggregationRepository` repositories implement this interface.

About LevelDB

LevelDB is a lightweight and embeddable key-value storage library. It allows Camel to provide persistence for various Camel features, such as the Aggregator.

You can find more information about LevelDB at its website: <http://github.com/google/leveldb>.

Now we'll look at how to use LevelDB as a persistent repository.

USING CAMEL-LEVELDB

To demonstrate how to use LevelDB with the Aggregator, we'll return to the *ABC* example. In essence, all you need to do is instruct the Aggregator to use `LevelDBAggregationRepository` as its repository.

First, though, you must set up LevelDB, which is done as follows:

```
AggregationRepository myRepo = new
    LevelDBAggregationRepository("myrepo",
"data/myrepo.dat");
```

Or, in XML, you do this:

```
<bean id="myRepo"
    class="org.apache.camel.component.leveldb.LevelDBAggregationRepository">
    <property name="repositoryName" value="myrepo"/>
    <property name="persistentFileName"
    value="data/myrepo.dat"/>
</bean>
```

As you can see, this creates a new instance of `LevelDBAggregationRepository` and provides two parameters: the repository name, which is a symbolic name, and the physical filename to use as persistent storage. The repository name must

be specified because you can have multiple repositories in the same file.

TIP You can find information about the additional supported options for the LevelDB component at the Camel website:
<http://camel.apache.org/leveldb>.

To use `LevelDBAggregationRepository` in the Camel route, you can instruct the Aggregator to use it, as shown in the following listing.

Listing 5.2 Using LevelDB with Aggregator in Java DSL

```
AggregationRepository myRepo =
    new LevelDBAggregationRepository("myrepo",
"data/myrepo.dat");

from("file:///target/inbox")
    .log("Consuming ${file:name}")
    .convertBodyTo(String.class)
    .aggregate(constant(true), new MyAggregationStrategy())
        .aggregationRepository(myRepo)
        .completionSize(3)
        .log("Sending out ${body}")
        .to("mock:result");
```

The next listing shows the same example in XML.

Listing 5.3 Using LevelDB with Aggregator in XML

```
<bean id="myAggregationStrategy"
      class="camelinaction.MyAggregationStrategy"/>
<bean id="myRepo" ①
```

①

LevelDB persistent repository

```
class="org.apache.camel.component.leveldb.LevelDBAggregationRepository">
```

```

<property name="repositoryName" value="myrepo"/>
    <property name="persistentFileName"
value="data/myrepo.dat"/>
</bean>

<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file://target/inbox"/>
        <log message="Consuming ${file:name}"/>
        <convertBodyTo type="java.lang.String"/>
        <aggregate strategyRef="myAggregationStrategy"
completionSize="3"
            aggregationRepositoryRef="myRepo"

```

As you can see, a Spring bean tag is defined with the ID `myRepo` **①**, which sets up the persistent AggregationRepository. The name for the repository and the filename are configured as properties on the bean tag. In the Camel route, you then refer to this repository by using the `aggregationRepositoryRef` attribute on the aggregate tag.

RUNNING THE EXAMPLE

The book's source code contains this example in the `chapter5/aggregator` directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=AggregateABCLevelDBTest
mvn test -Dtest=SpringAggregateABCLevelDBTest
```

To demonstrate how the persistence store works, the example will start up and run for 20 seconds. In that time, you can copy files in the `target/inbox` directory and have those files consumed and aggregated. On every third file, the Aggregator will complete and publish a message.

The example displays instructions on the console about how to do this:

```
Copy 3 files to target/inbox to trigger the completion
Files to copy:
  copy src/test/resources/a.txt target/inbox
  copy src/test/resources/b.txt target/inbox
  copy src/test/resources/c.txt target/inbox
Sleeping for 20 seconds
You can let the test terminate (or press ctrl + c) and then start it again
Which should let you be able to resume.
```

For instance, if you copy the first two files and then let the example terminate, you'll see the following:

```
cd chapter5/aggregator
chapter5/aggregator$ cp src/test/resources/a.txt
target/inbox
chapter5/aggregator$ cp src/test/resources/b.txt
target/inbox
```

The console should indicate that it consumed two files and was shut down:

```
2017-05-07 12:29:33,714 [ #1 - file://target/inbox]
  INFO route1 - Consuming file a.txt
2017-05-07 12:29:35,235 [ #1 - file://target/inbox]
  INFO route1 - Consuming file b.txt
...
2017-05-07 12:29:43,224 [ main]
  INFO DefaultCamelContext - Apache Camel 2.20.1
(CamelContext:
  camel-1) is shutdown in 0.022 seconds
```

The next time you start the example, you can resume where you left off, and copy the last file:

```
chapter5/aggregator$ cp src/test/resources/c.txt
target/inbox
```

Then the Aggregator should complete and publish the message:

```
2017-05-07 12:40:35,069 [ main]
  INFO LevelDBAggregationRepository - On startup there
are 1
```

```
aggregate exchanges (not completed) in repository:  
myrepo  
...  
2017-05-07 12:40:38,589 [ #1 - file://target/inbox]  
    INFO route1 - Consuming file c.txt  
2017-05-07 12:40:38,606 [ #1 - file://target/inbox]  
    INFO route1 - Sending out ABC
```

Notice that it logs on startup the number of exchanges that are in the persistent repository. This example has one existing Exchange on startup.

Now you've seen the persistent Aggregator in action. Let's move on to look at using recovery with the Aggregator, which ensures that published messages can be safely recovered and be routed in a transactional way.

5.2.4 USING RECOVERY WITH THE AGGREGATOR

The examples covered in the previous section focused on ensuring that messages are persisted during aggregation. But there's another way messages may be lost: messages that have been published (sent out) from the Aggregator could fail during routing as well.

To remedy this problem, you could use one of these two approaches:

- *Camel error handlers*—These provide redelivery and dead letter channel capabilities. We cover them in chapter 11.
- `RecoverableAggregationRepository`—This interface *extends* `AggregationRepository` and offers the recovery, redelivery, and dead letter channel features.

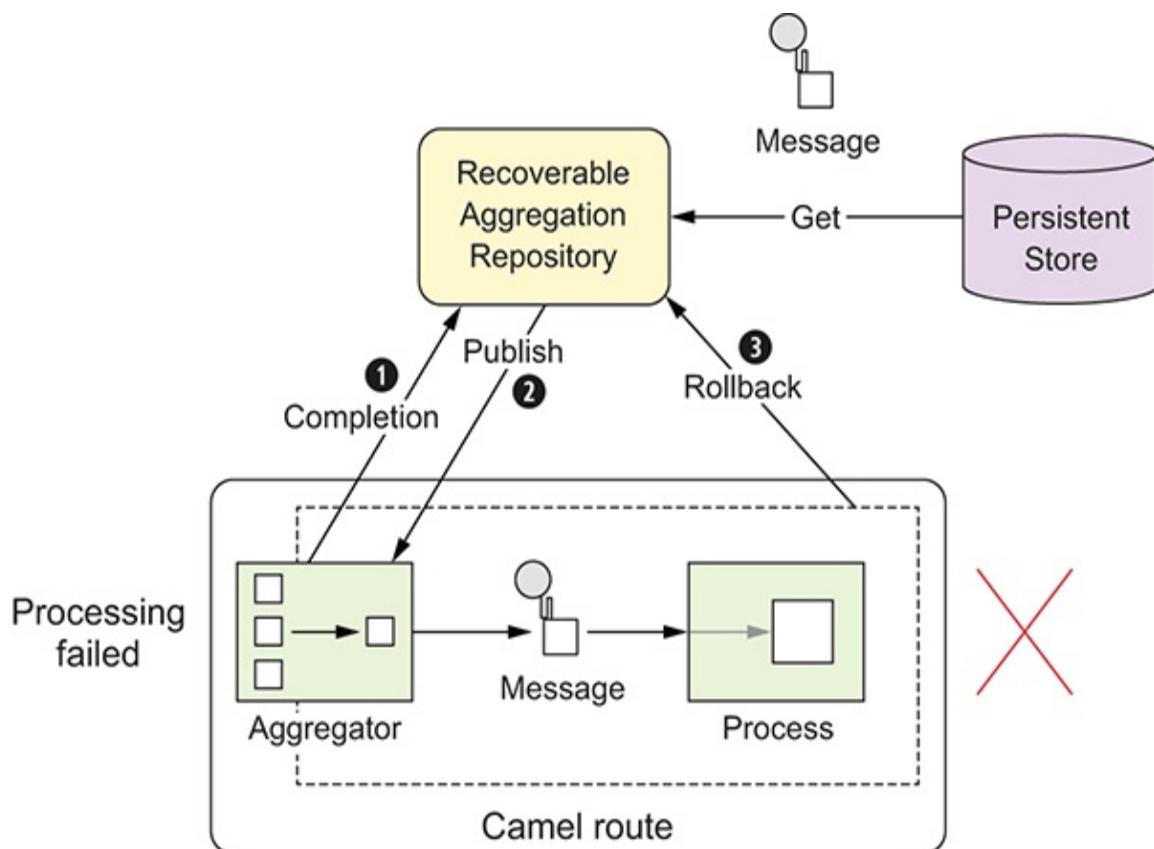
Camel error handlers aren't tightly coupled with the Aggregator, so message handling is in the hands of the error handler. If a message repeatedly fails, the error handler can deal with this only by retrying or eventually giving up and moving the message to a dead letter channel.

`RecoverableAggregationRepository`, on the other hand, is

tightly integrated into the Aggregator, which allows additional benefits such as using the persistence store for recovery and offering transactional capabilities. It ensures that published messages that fail will be recovered and redelivered. You can think of this as what a JMS broker, such as Apache ActiveMQ, can do by bumping failed messages back up on the JMS queue for redelivery.

UNDERSTANDING RECOVERY

To better understand how recovery works, consider the next two figures. [Figure 5.4](#) shows what happens when an aggregated message is being published for the first time and the message fails during processing. This could also be the situation when a server crashes while processing the message.



[Figure 5.4](#) An aggregated message is completed ①, it's published from the Aggregator ②, and processing fails ③, so the message is rolled back.

An aggregated message is complete, so the Aggregator signals ①

this to the RecoverableAggregationRepository, which fetches the aggregated message to be published ②. The message is then routed in Camel—but suppose it fails during routing ③? A signal is sent from the Aggregator to RecoverableAggregationRepository, which can act accordingly.

Now imagine the same message is recovered and redelivered, as shown in [figure 5.5](#).

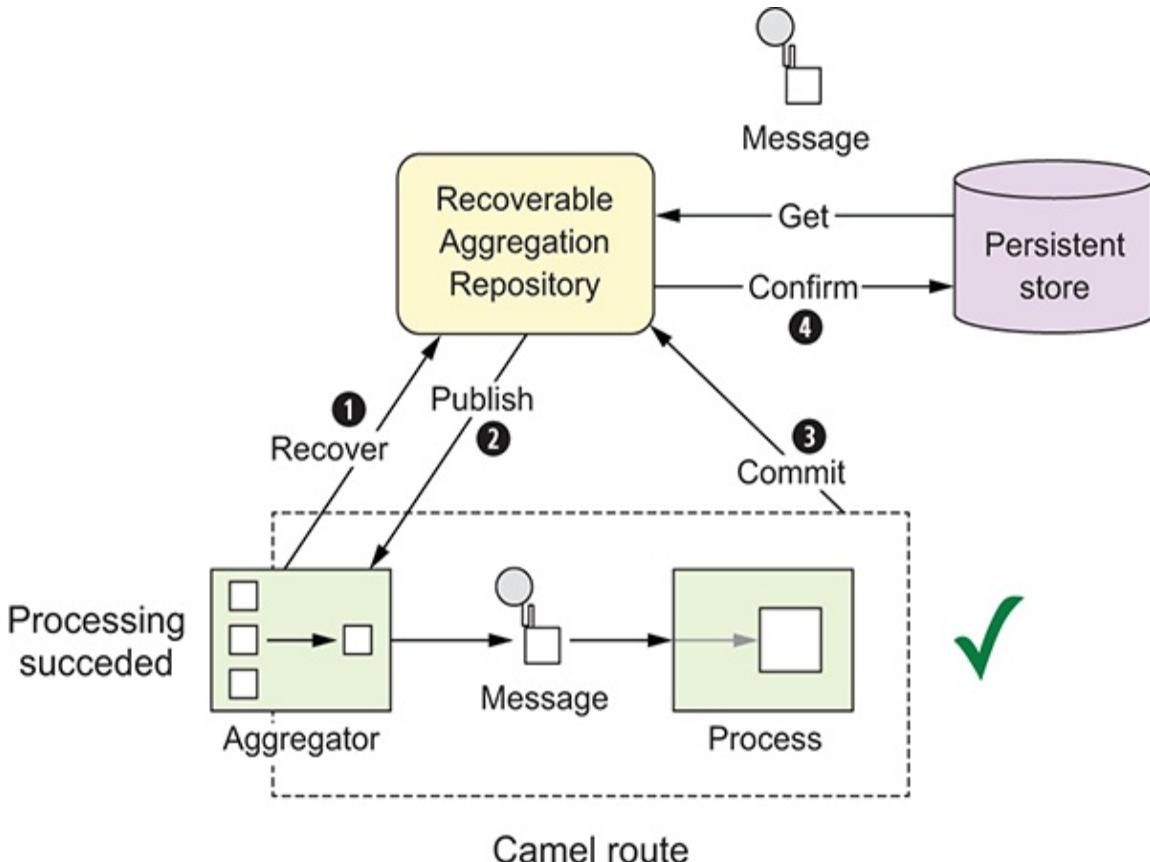


Figure 5.5 The Aggregator recovers failed messages ①, which are published again ②, and this time the messages completed ③ successfully ④.

The Aggregator uses a background task, which runs at regular intervals (use `setRecoveryInterval` on the `RecoverableAggregationRepository` to specify this), to scan for previously published messages to be recovered ①. Any such messages will be republished ②, and the message will be routed again. This time, the message could be processed successfully, which lets the Aggregator issue a commit ③. The repository confirms the message ④, ensuring that it won't be recovered on

subsequent scans.

NOTE The transactional behavior provided by `RecoverableAggregationRepository` isn't based on Spring's `TransactionManager` (which we cover in chapter 12). The transactional behavior is based on LevelDB's own transaction mechanism (because we're using the `LevelDBAggregationRepository`).

RUNNING THE EXAMPLE

The book's source code contains this example in the `chapter5/aggregator` directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=AggregateABCRecoverTest  
mvn test -Dtest=SpringAggregateABCRecoverTest
```

The example is constructed to fail when processing the published messages, no matter what. Eventually, you'll have to move the message to a dead letter channel.

To use recovery with routes in the Java DSL, you have to set up `LevelDBAggregationRepository` as shown here:

```
LevelDBAggregationRepository levelDB = new  
    LevelDBAggregationRepository("myrepo",  
"data/myrepo.dat");  
levelDB.setUseRecovery(true);  
levelDB.setMaximumRedeliveries(4);  
levelDB.setDeadLetterUri("mock:dead");  
levelDB.setRecoveryInterval(3000);
```

In XML, you can set this up as a `<bean>` tag, as follows:

```
<bean id="myRepo"  
  
class="org.apache.camel.component.leveldb.LevelDBAggregatio  
nRepository">  
    <property name="repositoryName" value="myrepo"/>  
    <property name="persistentFileName"
```

```

    value="data/myrepo.dat"/>
    <property name="useRecovery" value="true"/>
    <property name="recoveryInterval" value="3000"/>
    <property name="maximumRedeliveries" value="4"/>
    <property name="deadLetterUri" value="mock:dead"/>
</bean>

```

The options may make sense as you read them now, but you'll revisit them in [table 5.7](#). In this example, the Aggregator will check for messages to be recovered every 3 seconds. To avoid a message being repeatedly recovered, the maximum redeliveries are set to 4. After four failed recovery attempts, the message is exhausted and moved to the dead letter channel. If you omit the maximum redeliveries option, Camel will keep recovering failed messages forever until they can be processed successfully.

If you run the example, you'll notice that the console outputs the failures as stack traces, and at the end you'll see a `WARN` entry that indicates the message has been moved to the dead letter channel:

```

2017-05-07 22:28:18,997 [- AggregateRecoverChecker]
    WARN AggregateProcessor - The recovered exchange is
exhausted after 4
    attempts, will now be moved to dead letter channel:
mock:dead

```

We encourage you to try this example and read the comments in the source code to better understand how this works.

The preceding log output identifies the number of redelivery attempts, but how does Camel know this? Camel stores this information on the Exchange. [Table 5.6](#) reveals where this information is stored.

Table 5.6 Headers on Exchange related to redelivery

Header	Type	Description
Exchange.REDELIVERY_COUNTER	int	The current redelivery attempt. The counter starts with the value of 1.
Exchange.REDELIVERY_MAX_COUNTER	int	The maximum redelivery attempts that will be made.

Exchange.REDELIVERY_ERED	boolean	Whether this Exchange is being redelivered.
Exchange.REDELIVERY_EXHAUSTED	boolean	Whether this Exchange has attempted all redeliveries and still failed (also known as being exhausted).
Exchange.REDELIVERY_DELAY	long	Delay in milliseconds before scheduling redelivery.

The information in table 5.6 is available only when Camel performs a recovery. These headers are absent on the regular first attempt. It's only when a recovery is triggered that these headers are set on the Exchange.

Table 5.7 lists the options for `RecoverableAggregationRepository` that are related to recovery.

Table 5.7 `RecoverableAggregationRepository` configuration options related to recovery

Option	Default	Description
useRecovery	true	Whether recovery is enabled.
recoveryInterval	50000	How often the recovery background tasks are executed. The value is in milliseconds.
deadLetter		An optional dead letter channel, where published messages that are exhausted should be sent. This is similar to the <code>DeadLetterChannel</code> error handler, which we cover in chapter 11. This option is disabled by default. When in use, the <code>maximumRedeliveries</code> option must be

Uri	configured as well.
maxi mu mR ed el iv er ies	A limit that defines when published messages that repeatedly fail are considered exhausted and should be moved to the dead letter URI. This option is disabled by default.

We won't go into more detail regarding the options in table 5.7, as we've already covered an example using them.

This concludes our extensive coverage of the sophisticated and probably most complex EIP implemented in Camel—the Aggregator. In the next section, we'll look at the Splitter pattern.

5.3 The Splitter EIP

Messages passing through an integration solution may consist of multiple elements, such as an order, which typically consists of more than a single line item. Each line in the order may need to be handled differently, so you need an approach that processes the complete order, treating each line item individually. The solution to this problem is the Splitter EIP, illustrated in figure 5.6.



Figure 5.6 The Splitter breaks the incoming message into a series of individual messages.

In this section, we'll teach you all you need to know about the Splitter. You'll start with a simple example and move on from there.

5.3.1 USING THE SPLITTER

Using the Splitter in Camel is straightforward, so let's try a basic

example that will split one message into three messages, each containing one of the letters *A*, *B*, and *C*. The following listing shows the example using a Java DSL–based Camel route and a unit test.

Listing 5.4 A basic example of the Splitter EIP

```
public class SplitterABCTest extends CamelTestSupport {  
    public void testSplitABC() throws Exception {  
        MockEndpoint mock = getMockEndpoint("mock:split");  
        mock.expectedBodiesReceived("A", "B", "C");  
        List<String> body = new ArrayList<String>();  
        body.add("A");  
        body.add("B");  
        body.add("C");  
        template.sendBody("direct:start", body);  
        assertMockEndpointsSatisfied();  
    }  
    protected RouteBuilder createRouteBuilder() throws  
Exception {  
        return new RouteBuilder() {  
            public void configure() throws Exception {  
                from("direct:start")  
                    .split(body()) ②  
                    .log("Split line ${body}")  
                    .to("mock:split");  
            }  
        };  
    }  
}
```

1

Splits incoming message body

```
.log("Split line ${body}")  
.to("mock:split");  
}  
};  
}  
}
```

The test method sets up a mock endpoint that expects three messages to arrive, in the order *A*, *B*, and *C*. Then you construct a single combined message body that consists of a `List` of `Strings` containing the three letters. The Camel route will use the Splitter EIP to split up the message body **1**.

If you run this test, the console should log the three messages, as follows:

```
INFO route1 - Split line A  
INFO route1 - Split line B  
INFO route1 - Split line C
```

When using the Splitter EIP in XML, you have to do this differently because the Splitter uses an Expression to return what is to be split.

In the Java DSL, you defined the Expression shown in bold:

```
.split(body())
```

Here, body is a method available on RouteBuilder, which returns an org.apache.camel.Expression instance. In XML, you need to do this as shown in bold:

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
  <route>  
    <from uri="direct:start"/>  
    <split>  
      <simple>${body}</simple>  
      <log message="Split line ${body}" />  
      <to uri="mock:split"/>  
    </split>  
  </route>  
</camelContext>
```

In XML, you use Camel's expression language, known as Simple (discussed in appendix A), to tell the Splitter that it should split the message body.

The book's source code contains this example in the chapter5/splitter directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=SplitterABCTest  
mvn test -Dtest=SpringSplitterABCTest
```

Now you've seen the Splitter in action. To know how to tell Camel what it should split, you need to understand how it works.

How THE SPLITTER WORKS

The Splitter works something like a big iterator that iterates through something and processes each entry. The sequence diagram in figure 5.7 shows more details about how this *big iterator* works.

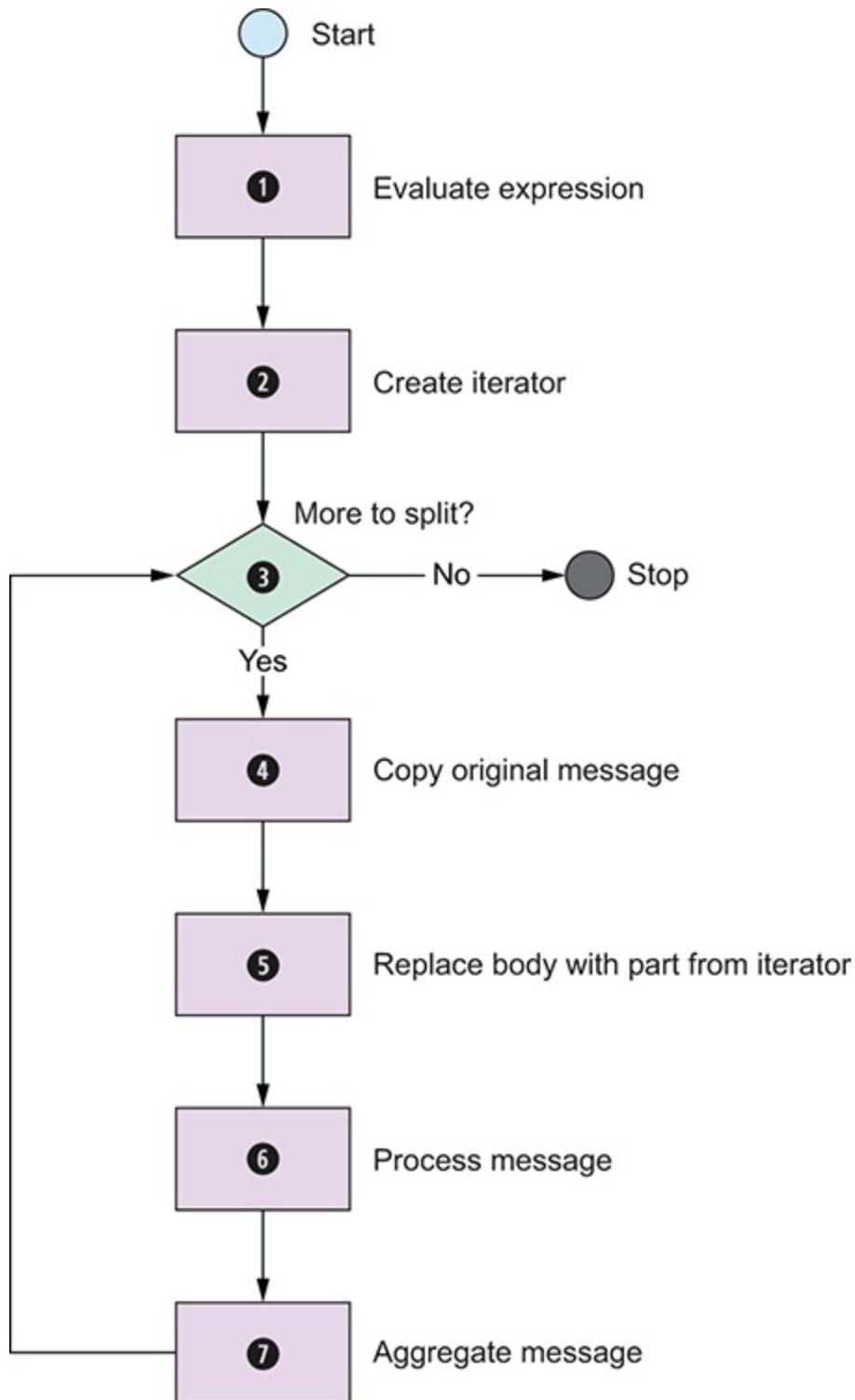


Figure 5.7 A sequence diagram showing how the Splitter works internally, by using an iterator to iterate through the message and process each entry

When working with the Splitter, you have to configure an Expression, which is evaluated ① when a message arrives. In listing 5.4, the evaluation returned the message body. The result from the evaluation is used to create java.util.Iterator ②.

What can be iterated?

When Camel creates the iterator ②, it supports a range of types. Camel knows how to iterate through the following types: Collection, Iterator, Iterable, org.w3c.dom, NodeList, String (with entries separated by commas) and arrays. Any other type will be iterated once.

Then the Splitter uses the iterator ③ until there's no more data. Each message to be sent out of the iterator is a copy of the original message ④, which has had its message body replaced (org.apache.camel.Message.setBody is called) with the part from the iterator ⑤. In listing 5.4, there would be three parts: each of the letters *A*, *B*, and *C*. The message to be sent out is then processed ⑥, and when the processing is done, the message may be aggregated ⑦ (more about this in section 5.3.4).

The Splitter will decorate each message it sends out with properties on the Exchange, which are listed in table 5.8.

Table 5.8 Properties on the Exchange related to the Splitter EIP

Property	Type	Description
Exchange.SP_LIT_INDEX	int	The index for the current message being processed. The index is zero-based.
Exchange.SP_LIT_SIZE	int	The total number of messages that the original message has been split into. Note that this information isn't available in streaming mode

		(see section 5.3.3 for more details about streaming).
Exchange.SP LIT_COMPLETE	bo ol ea n	Whether or not this is the last message being processed.

You may find yourself needing more power to do the splitting, such as to dictate exactly how a message should be split. And what better power is there than Java? By using Java code, you have the ultimate control and can tackle any situation.

5.3.2 USING BEANS FOR SPLITTING

Suppose you need to split messages that contain complex payloads. And suppose the message payload is a `Customer` object containing a list of `Departments`, and you want to split by `Department`, as illustrated in [figure 5.8](#).

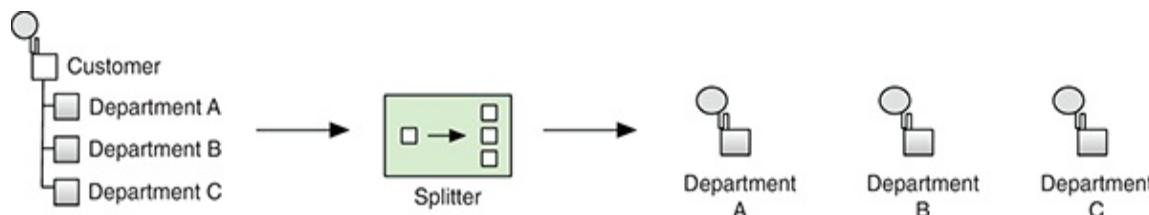


Figure 5.8 Splitting a complex message into submessages by department

The `Customer` object is a simple bean containing the following information (getter and setter methods omitted):

```

public class Customer {
    private int id;
    private String name;
    private List<Department> departments;
}
  
```

The `Department` object is simple as well:

```

public class Department {
    private int id;
    private String address;
    private String zip;
    private String country;
}
  
```

You may wonder why you can't split the message as in the previous example, using `split(body())`. The reason is that the message payload (the message body) isn't a `List`, but a `Customer` object. Therefore, you need to tell Camel how to split, which you do as follows:

```
public class CustomerService {  
    public List<Department> splitDepartments(Customer  
customer) {  
        return customer.getDepartments();  
    }  
}
```

The `splitDepartments` method returns a `List` of `Department` objects, which is what you want to split by.

In the Java DSL, you can use the `CustomerService` bean for splitting by telling Camel to invoke the `splitDepartments` method. This is done by using the `method` call expression, as shown in bold:

```
public void configure() throws Exception {  
    from("direct:start")  
        .split().method(CustomerService.class,  
"splitDepartments")  
            .to("log:split")  
            .to("mock:split");  
}
```

In XML, you'd have to declare the `CustomerService` in a `<bean>` tag, as follows:

```
<bean id="customerService"  
class="camelaction.CustomerService"/>  
  
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
    <route>  
        <from uri="direct:start"/>  
        <split>  
            <method bean="customerService"  
method="splitDepartments"/>  
                <to uri="log:split"/>  
                <to uri="mock:split"/>  
        </split>
```

```
</route>  
</camelContext>
```

The book's source code contains this example in the chapter5/splitter directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=SplitterBeanTest  
mvn test -Dtest=SpringSplitterBeanTest
```

The logic in the `splitDepartments` method is simple, but it shows you how to use a method on a bean to do the splitting. In your use cases, you may need more complex logic.

TIP The logic in the `splitDepartments` method seems trivial, and it's possible to use Camel's expression language (Simple) to invoke methods on the message body. In Java DSL, you could define the route as

follows: `.split().simple("${body.departments}")`. In XML you'd use the `<simple>` tag instead of the `<method>` tag:
`<simple>${body.departments}</simple>`.

The Splitter will usually operate on messages that are loaded into memory. But in some situations, the messages are so big that it's not feasible to have the entire message in memory at once.

5.3.3 SPLITTING BIG MESSAGES

Rider Auto Parts has an ERP system that contains inventory information from all its suppliers. To keep the inventory updated, each supplier must submit updates to Rider Auto Parts. Some suppliers do this once a day, using good old-fashioned files as a means of transport. Those files could be large, so you have to split those files without loading the entire file into memory.

You can do that by using streams, which allow you to read on demand from a stream of data. This resolves the memory issue, because you can read in a chunk of data, process the data, read in another chunk, process the data, and so on.

Figure 5.9 shows the flow of the application used by Auto Rider Parts to pick up the files from the suppliers and update the inventory.

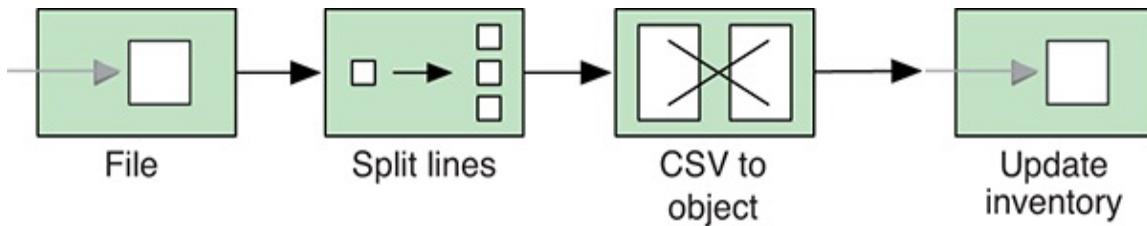


Figure 5.9 A route that picks up incoming files, splits them, and transforms them so they're ready for updating the inventory in the ERP system

We'll revisit this example again in chapter 13, and cover it in much greater detail when we cover concurrency.

Implementing the route outlined in [figure 5.9](#) is easy to do in Camel, as shown in the following listing.

Listing 5.5 Splitting big files by using streaming mode

```
public void configure() throws Exception {  
    from("file:target/inventory")  
        .log("Starting to process big file:  
${header.CamelFileName}")  
        .split(body().tokenize("\n")).streaming() ②  
}
```

1

Splits file using streaming mode

```
    .bean(InventoryService.class, "csvToObject")  
    .to("direct:update")  
    .end() ②
```

2

Denotes where the splitting route ends

```
    .log("Done processing big file:  
${header.CamelFileName}");  
  
    from("direct:update")  
        .bean(InventoryService.class, "updateInventory");
```

```
}
```

As you can see, all you have to do is enable streaming mode by using `.streaming` ❶. This tells Camel to not load the entire payload into memory, but instead to iterate the payload in a streaming fashion. Also notice the use of `end` ❷ to indicate the end of the splitting route. The `end` in the Java DSL is the equivalent of the end tag `</split>` when using XML.

In XML, you enable streaming by using the `streaming` attribute on the `<split>` tag, as shown in the following listing.

Listing 5.6 Splitting big files by using streaming mode in XML

```
<camelContext
    xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file:target/inventory"/>
        <log message="Processing big file:
${header.CamelFileName}"/>
        <split streaming="true">
            <tokenize token="\n"/>
            <bean beanType="camelaction.InventoryService"
                  method="csvToObject"/>
            <to uri="direct:update"/>
        </split>
        <log message="Done processing big file:
${header.CamelFileName}"/>
    </route>
    <route>
        <from uri="direct:update"/>
        <bean beanType="camelaction.InventoryService"
              method="updateInventory"/>
    </route>
</camelContext>
```

You may have noticed in listings 5.5 and 5.6 that the files are split by using a tokenizer. The *tokenizer* is a powerful feature that works well with streaming. It uses `java.util.Scanner` under the hood, which reads chunks of data into memory. A token must be provided to indicate the boundaries of the chunks. In the preceding code, you use a newline (`\n`) as the token. In this example, the `Scanner` will read the file into memory on a line-by-

line basis, resulting in low memory consumption.

NOTE When using streaming mode, be sure the message you're splitting can be split into well-known chunks that can be iterated. You can use the tokenizer (the `java.util.Scanner` used implements `Iterator`) or convert the message body to a type that can be iterated, such as an `Iterator`.

The Splitter EIP in Camel includes an aggregation feature that lets you recombine split messages into single outbound messages, while they're being routed.

5.3.4 AGGREGATING SPLIT MESSAGES

Being able to split and aggregate messages again is a powerful mechanism. You could use this to split an order into individual order lines, process them, and then recombine them into a single outgoing message. This pattern is known as the Composed Message Processor, which we briefly touched on in section 5.1. It's shown in [figure 5.1](#).

The Camel Splitter provides a built-in aggregator, which makes it even easier to aggregate split messages back into single outgoing messages. [Figure 5.10](#) illustrates this principle, with the help of the *ABC* message example.

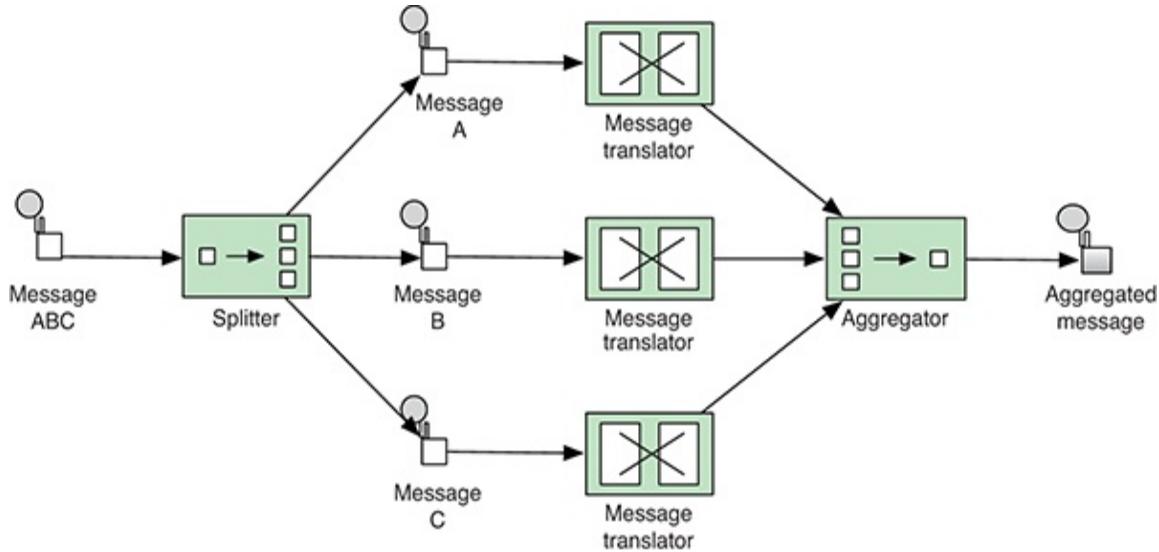


Figure 5.10 The Splitter has a built-in aggregator that can recombine split messages into a combined outgoing message.

Suppose you want to translate each of the *A*, *B*, and *C* messages into a phrase, and have all the phrases combined into a single message again. You can easily do this with the Splitter; all you need to provide is the logic that combines the messages. This logic is created using an `AggregationStrategy` implementation.

Implementing the Camel route outlined in [figure 5.10](#) can be done as follows in the Java DSL. The configuration of the `AggregationStrategy` is shown in bold:

```

from("direct:start")
    .split(body(), new MyAggregationStrategy())
        .log("Split line ${body}")
        .bean(WordTranslateBean.class)
        .to("mock:split")
    .end()
    .log("Aggregated ${body}")
    .to("mock:result");
  
```

In XML, you have to declare `AggregationStrategy` as a `<bean>` tag, as shown in bold:

```

<bean id="translate"
  class="camelaction.WordTranslateBean"/>
<bean id="myAggregationStrategy"
  class="camelaction.MyAggregationStrategy"/>
  
```

```

<camelContext
    xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <split strategyRef="myAggregationStrategy">
            <simple>body</simple>
            <log message="Split line ${body}" />
            <bean ref="translate"/>
            <to uri="mock:split"/>
        </split>
        <log message="Aggregated ${body}" />
        <to uri="mock:result"/>
    </route>
</camelContext>

```

To combine the split messages back into a single combined message, you use AggregationStrategy, as shown in the following listing.

Listing 5.7 Combining split messages back into a single outgoing message

```

public class MyAggregationStrategy implements
AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange,
Exchange newExchange) {
        if (oldExchange == null) {
            return newExchange;
        }
        String body =
newExchange.getIn().getBody(String.class).trim();
        String existing =
oldExchange.getIn().getBody(String.class).trim();
        oldExchange.getIn().setBody(existing + "+" + body);
        return oldExchange;
    }
}

```

As you can see, you combine the messages into a single string body, with individual phrases (from the message bodies) being separated with + signs.

The source code for the book contains this example in the chapter5/splitter directory. You can run it using the following Maven goals:

```
mvn test -Dtest=SplitterAggregateABCTest  
mvn test -Dtest=SpringSplitterAggregateABCTest
```

The example uses the three phrases: *Aggregated Camel rocks*, *Hi mom*, and *Yes it works*. When you run the example, you'll see the console output the aggregated message at the end:

```
INFO route1 - Split line A  
INFO route1 - Split line B  
INFO route1 - Split line C  
INFO route1 - Aggregated Camel rocks+Hi mom+Yes it works
```

Before we wrap up our coverage of the Splitter, let's take a look at what happens if one of the split messages fails with an exception.

5.3.5 WHEN ERRORS OCCUR DURING SPLITTING

The Splitter processes messages, and those messages can fail when business logic throws an exception. Camel's error handling is active during the splitting, so the errors you have to deal with in the Splitter are errors that couldn't be handled by the error-handling rules you defined.

You have two choices for handling errors with the Splitter:

- *Stop*—The Splitter will split and process each message in sequence. Suppose the second message failed. In this situation, you could either immediately stop and let the exception propagate back, or you could continue splitting the remainder of the messages, and let the exception propagate back at the end (default behavior).
- *Aggregate*—You could handle the exception in `AggregationStrategy` and decide whether the exception should be propagated back.

Let's look into the choices.

USING STOPONEXCEPTION

The first solution requires you to configure the `stopOnException` option on the Splitter as follows:

```
from("direct:start")
    .split(body(), new MyAggregationStrategy())
        .stopOnException()
        .log("Split line ${body}")
        .bean(WordTranslateBean.class)
        .to("mock:split")
    .end()
    .log("Aggregated ${body}")
    .to("mock:result");
```

In XML, you use the `stopOnException` attribute on the `<split>` tag, as follows:

```
<split strategyRef="myAggregationStrategy"
stopOnException="true">
```

The book's source code contains this example in the `chapter5/splitter` directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=SplitterStopOnExceptionABCTest
mvn test -Dtest=SpringSplitterStopOnExceptionABCTest
```

The second option is to handle exceptions from the split messages in `AggregationStrategy`.

HANDLING EXCEPTIONS USING AGGREGATIONSTRATEGY

`AggregationStrategy` allows you to handle the exception by either ignoring it or letting it be propagated back. Here's how you could ignore the exception.

[Listing 5.8](#) Handling an exception by ignoring it

```
public class MyIgnoreFailureAggregationStrategy
    implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange,
Exchange newExchange) {
        if (newExchange.getException() != null) {
            return oldExchange; ❶
        }
    }
}
```

1

Ignores the exception

```
    }
    if (oldExchange == null) {
        return newExchange;
    }
    String body =
newExchange.getIn().getBody(String.class);
    String existing =
oldExchange.getIn().getBody(String.class);
    oldExchange.getIn().setBody(existing + "+" + body);
    return oldExchange;
}
}
```

When handling exceptions in `AggregationStrategy`, you can detect whether an exception occurred by checking the `getException` method from the `newExchange` parameter. The preceding example ignores the exception by returning `oldExchange` ①.

If you want to propagate back the exception, you need to keep it stored on the aggregated exception, as shown in the following listing.

[Listing 5.9 Propagating back an exception](#)

```
public class MyPropagateFailureAggregationStrategy
    implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange,
Exchange newExchange) {
        if (newExchange.getException() != null) {
            if (oldExchange == null) {
                return newExchange;
            } else {
                oldExchange.setException(
                    newExchange.getException());
            }
        }
    }
}
```

1

Propagates exception

```

        return oldExchange;
    }
}
if (oldExchange == null) {
    return newExchange;
}
String body =
newExchange.getIn().getBody(String.class);
String existing =
oldExchange.getIn().getBody(String.class);
oldExchange.getIn().setBody(existing + "+" + body);
return oldExchange;
}
}

```

As you can see, it requires a bit more work to keep the exception. On the first invocation of the aggregate method, the `oldExchange` parameter is `null`, and you return the `newExchange` (which has the exception). Otherwise, you must transfer the exception to `oldExchange` ①.

WARNING When using a custom `AggregationStrategy` with the Splitter, it's important to know that you're responsible for handling exceptions. If you don't propagate the exception back, the Splitter will assume you've handled the exception and will ignore it.

The book's source code contains this example in the `chapter5/splitter` directory. You can run it by using the following Maven goals:

```

mvn test -Dtest=SplitterAggregateExceptionABCTest
mvn test -Dtest=SpringSplitterAggregateExceptionABCTest

```

Now you've learned all there is to know about the Splitter. Well, almost all. We'll revisit the Splitter in chapter 13 when we look at concurrency. In the next two sections, you'll see EIPs that support dynamic routing, starting with the Routing Slip pattern.

5.4 The Routing Slip EIP

At times, you need to route messages dynamically. For example, you may have an architecture that requires incoming messages to undergo a sequence of processing steps and business rule validations. Because the steps and validations vary widely, you can implement each step as a separate filter. The filter acts as a dynamic model to apply the business rule and validations.

This architecture could be implemented by using the Pipes and Filters EIP together with the Filter EIP. But as often happens with EIPs, there's a better way—in this case, the Routing Slip EIP. The Routing Slip acts as a dynamic router that dictates the next step a message should undergo. [Figure 5.11](#) shows this principle.

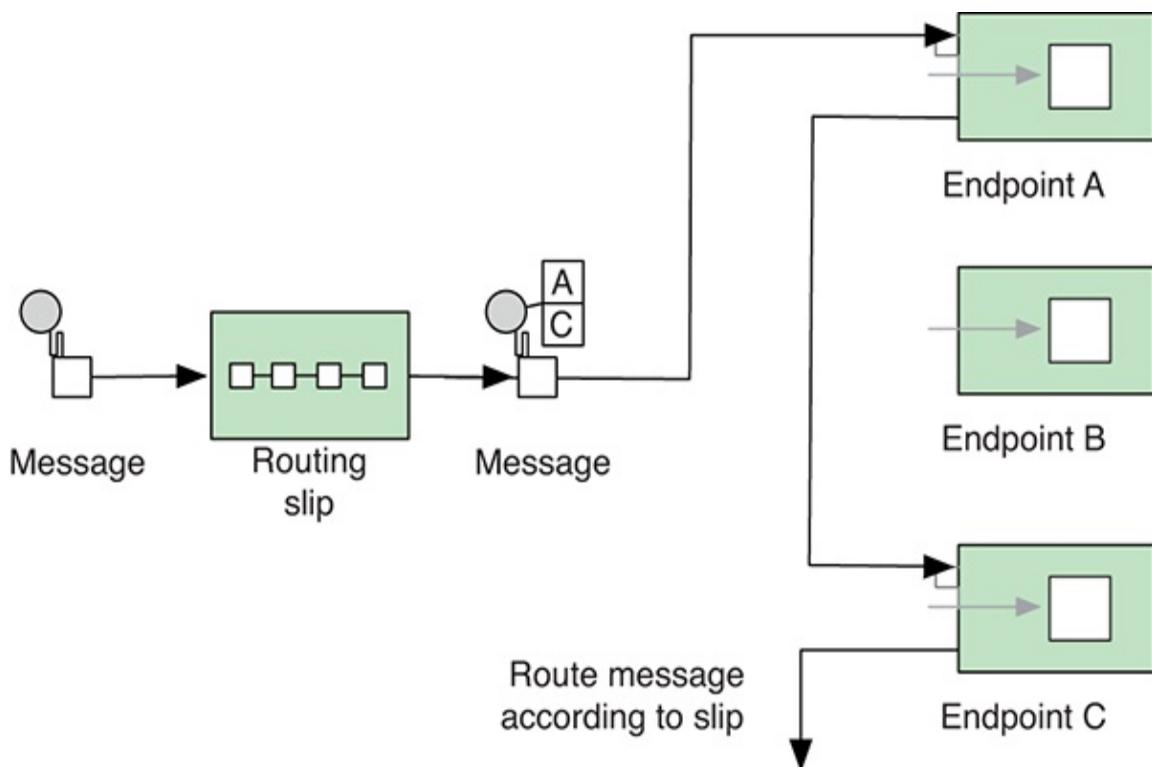


Figure 5.11 The incoming message has a slip attached that specifies the sequence of the processing steps. The Routing Slip EIP reads the slip and routes the message to the next endpoint in the list.

The Camel Routing Slip EIP requires a preexisting header or Expression as the attached slip. Either way, the initial slip must

be prepared before the message is sent to the Routing Slip EIP.

5.4.1 USING THE ROUTING SLIP EIP

We'll start with a simple example that shows how to use the Routing Slip EIP to perform the sequence outlined in [figure 5.11](#).

In the Java DSL, the route is as simple as this:

```
from("direct:start").routingSlip(header("mySlip"));
```

It's also easy in XML:

```
<route>
    <from uri="direct:start"/>
    <routingSlip>
        <header>mySlip</header>
    </routingSlip>
</route>
```

This example assumes that the incoming message contains the slip in the header with the key `mySlip`. The following test method shows how you should fill out the key:

```
public void testRoutingSlip() throws Exception {
    getMockEndpoint("mock:a").expectedMessageCount(1);
    getMockEndpoint("mock:b").expectedMessageCount(0);
    getMockEndpoint("mock:c").expectedMessageCount(1);
    template.sendBodyAndHeader("direct:start", "Hello
World",
                                "mySlip", "mock:a/mock:c");
    assertMockEndpointsSatisfied();
}
```

As you can see, the value of the key is the endpoint URIs separated by commas. The comma is the default delimiter, but the routing slip supports using custom delimiters. For example, to use a semicolon, you could do this:

```
from("direct:start").routingSlip(header("mySlip"), ";" );
```

And in XML, you'd do this:

```
<routingSlip uriDelimiter=";">
    <header>mySlip</header>
```

```
</routingSlip>
```

This example expects a preexisting header containing the routing slip. But what if the message doesn't contain such a header? In those situations, you have to compute the header in any way you like. In the next example, you'll look at how to compute the header by using a bean.

5.4.2 USING A BEAN TO COMPUTE THE ROUTING SLIP HEADER

To keep things simple, the logic to compute a header that contains two or three steps has been kept in a single method, as follows:

```
public class ComputeSlip {  
    public String compute(String body) {  
        String answer = "mock:a";  
        if (body.contains("Cool")) {  
            answer += ",mock:b";  
        }  
        answer += ",mock:c";  
        return answer;  
    }  
}
```

All you have to do now is use this bean to compute the header to be used as the routing slip.

In the Java DSL, you can use the method call expression to invoke the bean and set the header:

```
from("direct:start")  
    .setHeader("mySlip").method(ComputeSlip.class)  
    .routingSlip(header("mySlip"));
```

In XML, you can do it as follows:

```
<route>  
    <from uri="direct:start"/>  
    <setHeader headerName="mySlip">  
        <method beanType="camelaction.ComputeSlip"/>  
    </setHeader>  
    <routingSlip>  
        <header>mySlip</header>
```

```
</routingSlip>  
</route>
```

In this example, you use a method call expression to set a header that's then used by the routing slip. But you might want to skip the step of setting the header and instead use the expression directly.

5.4.3 USING AN EXPRESSION AS THE ROUTING SLIP

Instead of using a header expression, you can use any other Expression to generate the routing slip. For example, you could use a method call expression, as covered in the previous section. Here's how you'd do so with the Java DSL:

```
from("direct:start")  
    .routingSlip(method(ComputeSlip.class));
```

The equivalent XML is as follows:

```
<route>  
    <from uri="direct:start"/>  
    <routingSlip>  
        <method beanType="camelaction.ComputeSlip"/>  
    </routingSlip>  
</route>
```

Another way of using the Routing Slip EIP in Camel is to use beans and annotations.

5.4.4 USING @ROUTINGSLIP ANNOTATION

The `@RoutingSlip` annotation allows you to turn a regular bean method into the Routing Slip EIP. Let's go over an example.

Suppose you have the following `SlipBean`:

```
public class SlipBean {  
    @RoutingSlip  
    public String slip(String body) {  
        String answer = "mock:a";  
        if (body.contains("Cool")) {  
            answer += ",mock:b";  
        }  
    }  
}
```

```
        answer += ",mock:c";
        return answer;
    }
}
```

As you can see, all this does is annotate the `slip` method with `@RoutingSlip`. When Camel invokes the `slip` method, it detects the `@RoutingSlip` annotation and continues routing according to the Routing Slip EIP.

WARNING When using `@RoutingSlip`, it's important to not use `routingSlip` in the DSL at the same time. When using both, Camel will double up using the Routing Slip EIP, which isn't the intention. Instead, do as shown in the following example.

Notice that there's no mention of the routing slip in the DSL. The route is just invoking a bean:

```
from("direct:start").bean(SlipBean.class);
```

Here it is in the XML DSL:

```
<bean id="myBean" class="camelaction.SlipBean"/>

<route>
    <from uri="direct:start"/>
    <bean ref="myBean"/>
</route>
```

Why might you want to use this? Well, by using `@RoutingSlip` on a bean, it becomes more flexible in the sense that the bean is accessible using an endpoint URI. Any Camel client or route could easily send a message to the bean and have it continue being routed as a routing slip.

Messages can also be sent “by hand” to the bean by using `ProducerTemplate`. A template class, in general, is a utility class that simplifies access to an API—in this case, the `Producer` interface. For example, by using `ProducerTemplate`, you could send a message to the bean like this:

```
ProducerTemplate template = ...  
template.sendBody("bean:myBean", "Camel rocks");
```

That *Camel rocks* message would then be routed as a routing slip with the slip generated as the result of the `myBean` method invocation.

The source code for the book contains the examples we've covered in the `chapter5/routingslip` directory. You can try them by using the following Maven goals:

```
mvn test -Dtest=RoutingSlipSimpleTest  
mvn test -Dtest=SpringRoutingSlipSimpleTest  
mvn test -Dtest=RoutingSlipHeaderTest  
mvn test -Dtest=SpringRoutingSlipHeaderTest  
mvn test -Dtest=RoutingSlipTest  
mvn test -Dtest=SpringRoutingSlipTest  
mvn test -Dtest=RoutingSlipBeanTest  
mvn test -Dtest=SpringRoutingSlipBeanTest
```

You've now seen the Routing Slip EIP in action.

5.5 The Dynamic Router EIP

In the previous section, you learned that the Routing Slip pattern acts as a dynamic router. What's the difference between the Routing Slip and Dynamic Router EIPs? The difference is minimal: the Routing Slip needs to compute the slip up front, whereas the Dynamic Router will evaluate on the fly where the message should go next.

5.5.1 USING THE DYNAMIC ROUTER

Just like the Routing Slip, the Dynamic Router requires you to provide *logic*, which determines where the message should be routed. Such logic is easily implemented by using Java code, and in this code you have total freedom to determine where the message should go next. For example, you might query a database or a rules engine to compute where the message should go.

The following listing shows the Java bean used in the example.

Listing 5.10 Java bean deciding where the message should be routed next

```
public class DynamicRouterBean {  
    public String route(String body,  
                        @Header(Exchange.SLIP_ENDPOINT) String  
previous) { 1
```

1

Previous endpoint URI

```
        return whereToGo(body, previous);  
    }  
  
    private String whereToGo(String body, String previous)  
{  
        if (previous == null) {  
            return "mock://a";  
        } else if ("mock://a".equals(previous)) {  
            return "language://simple:Bye ${body}";  
        } else {  
            return null; 2
```

2

Ends Dynamic Router

```
    }  
}
```

The idea with the Dynamic Router is to let Camel keep invoking the `route` method until it returns `null`. The first time the `route` method is invoked, the `previous` parameter will be `null` **1**. On every subsequent invocation, the `previous` parameter contains the endpoint URI of the last step.

As you can see in the `whereToGo` method, you use this fact and return different URIs depending on the previous step. When the dynamic router is to end, you return `null` **2**. It's very important that the Dynamic Router must return `null` at some point—

otherwise, Camel will route the message forever.

Using the Dynamic Router from the Java DSL is easy to do:

```
from("direct:start")
    .dynamicRouter(method(DynamicRouterBean.class,
"route"))
    .to("mock:result");
```

The same route in XML is just as easy, as shown here:

```
<bean id="myDynamicRouter"
      class="camelinaction.DynamicRouterBean"/>

<camelContext
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <dynamicRouter>
      <method ref="myDynamicRouter" method="route"/>
    </dynamicRouter>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

The book's source code contains this example in the chapter5/dynamicrouter directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=DynamicRouterTest
mvn test -Dtest=SpringDynamicRouterTest
```

You can also use the Dynamic Router annotation.

5.5.2 USING THE @DYNAMICROUTER ANNOTATION

To demonstrate how to use the `@DynamicRouter` annotation, let's change the previous example to use the annotation instead. To do that, annotate the Java code from [listing 5.10](#) as follows:

```
@DynamicRouter
public String route(String body,
                     @Header(Exchange.SLIP_ENDPOINT)
String previous) {
    ...
}
```

```
}
```

The next step is to invoke the `route` method on the bean, as if it were a regular bean. That means you shouldn't use the Routing Slip EIP in the route, but use a bean instead.

In the Java DSL, this is done as follows:

```
from("direct:start")
    .bean(DynamicRouterBean.class, "route")
    .to("mock:result");
```

In XML, you likewise change the `<dynamicRouter>` to a `<bean>` tag:

```
<camelContext
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <bean ref="myDynamicRouter" method="route"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

WARNING When using `@DynamicRouter`, it's important to not use `dynamicRouter` in the DSL at the same time. Instead, do as shown in the preceding example.

The book's source code contains this example in the `chapter5/dynamicrouter` directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=DynamicRouterAnnotationTest
mvn test -Dtest=SpringDynamicRouterAnnotationTest
```

This concludes the coverage of the dynamic routing patterns. In the next section, you'll learn about Camel's built-in Load Balancer EIP, which is useful when an existing load-balancing solution isn't in place.

5.6 The Load Balancer EIP

You may already be familiar with the load-balancing concept in computing. *Load balancing* is a technique to distribute workload across computers or other resources “in order to get optimal resource utilization, maximize throughput, minimize response time, and avoid overload” (http://en.wikipedia.org/wiki/Load_balancer). This service can be provided either in the form of a hardware device or as a piece of software, such as the Load Balancer EIP in Camel.

NOTE The Load Balancer wasn’t covered in the EIP book, but will likely be added if there’s a second edition of the book.

In this section, we introduce the Load Balancer EIP by walking through an example. Then, in section 5.6.2, we present the various types of load balancers Camel offers out of the box. We focus on the failover type in section 5.6.3 and finally show how to build your own load balancer in section 5.6.4.

5.6.1 INTRODUCING THE LOAD BALANCER EIP

The Camel Load Balancer EIP is a processor that implements the `org.apache.camel.processor.loadbalancer.LoadBalancer` interface. `LoadBalancer` offers methods to add and remove processors that should participate in the load balancing.

By using processors instead of endpoints, the load balancer is capable of balancing anything you can define in your Camel routes. But, that said, you’ll most often balance across a number of remote services. Such an example is illustrated in [figure 5.12](#), where a Camel application needs to load balance across two services.

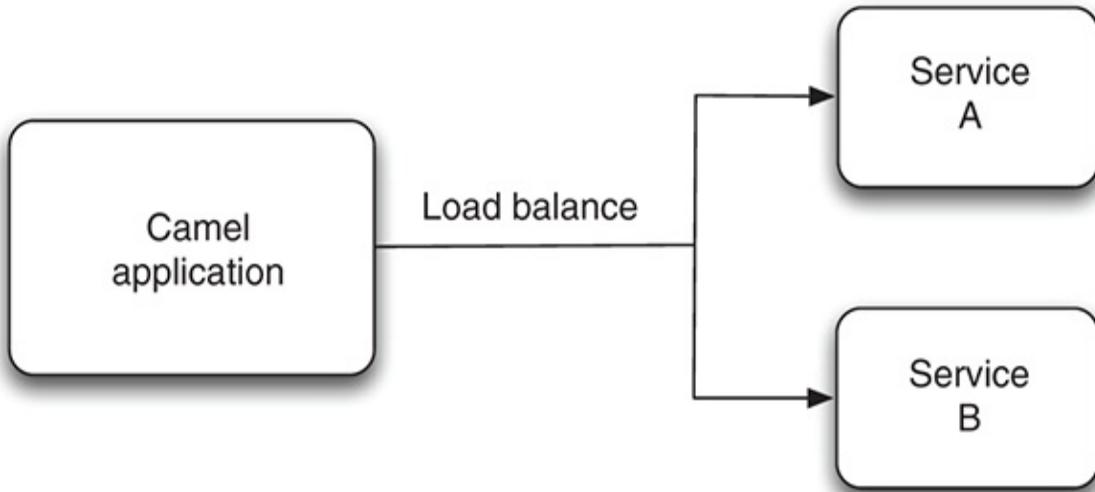


Figure 5.12 A Camel application load-balances across two services.

When using the Load Balancer EIP, you have to select a balancing strategy. A common and understandable strategy is to take turns among the services; this is known as the *round-robin strategy*. In section 5.6.2, we'll take a look at all the strategies Camel provides out of the box.

Let's look at how to use the Load Balancer with the round-robin strategy. Here's the Java DSL with the Load Balancer:

```

from("direct:start")
    .loadBalance().roundRobin()
        .to("seda:a").to("seda:b")
    .end();
from("seda:a")
    .log("A received: ${body}")
    .to("mock:a");
from("seda:b")
    .log("B received: ${body}")
    .to("mock:b");

```

The equivalent route in XML is as follows:

```

<route>
    <from uri="direct:start"/>
    <loadBalance>
        <roundRobin/>
        <to uri="seda:a"/>
        <to uri="seda:b"/>
    </loadBalance>

```

```
</route>
<route>
    <from uri="seda:a"/>
    <log message="A received: ${body}" />
    <to uri="mock:a"/>
</route>
<route>
    <from uri="seda:b"/>
    <log message="B received: ${body}" />
    <to uri="mock:b"/>
</route>
```

In this example, you use the SEDA component to simulate the remote services. In a real-life situation, the remote services could be a RESTful web service.

Suppose you start sending messages to the route. The first message would be sent to the seda:a endpoint, and the next would go to seda:b. The third message would start over and be sent to seda:a, and so forth.

The book's source code contains this example in the chapter5/loadbalancer directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=LoadBalancerTest
mvn test -Dtest=SpringLoadBalancerTest
```

If you run the example, the console will output something like this:

```
[Camel Thread 0 - seda://a] INFO route2 - A received: Hello
[Camel Thread 1 - seda://b] INFO route3 - B received: Camel
rocks
[Camel Thread 0 - seda://a] INFO route2 - A received: Cool
[Camel Thread 1 - seda://b] INFO route3 - B received: Bye
```

The next section reviews the various load-balancing strategies you can use with the Load Balancer EIP.

5.6.2 USING LOAD-BALANCING STRATEGIES

A load-balancing strategy dictates which processor should process an incoming message—it's up to each strategy how the

processor is chosen. Camel allows the six strategies listed in table 5.9.

Table 5.9 Load-balancing strategies provided by Camel

Strategy	Description
Random	Chooses a processor randomly.
Round-robin	Chooses a processor in a round-robin fashion, which spreads the load evenly. This is a classic and well-known strategy. We covered this in section 5.6.1.
Sticky	Uses an expression to calculate a correlation key that dictates the processor chosen. You can think of this as the session ID used in HTTP requests.
Topic	Sends the message to all processors. This is like sending to a JMS topic.
Failover	Retries using another processor. We cover this in section 5.6.3.
Custom	Uses your own custom strategy. This is covered in section 5.6.4.

The first four strategies in table 5.9 are easy to set up and use in Camel. For example, using the random strategy is just a matter of specifying it in the Java DSL:

```
from("direct:start")
    .loadBalance().random()
        .to("seda:a").to("seda:b")
    .end();
```

It's similar in XML:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <random/>
        <to uri="seda:a"/>
        <to uri="seda:b"/>
    </loadBalance>
</route>
```

The sticky strategy requires that you provide a correlation expression, which is used to calculate a hashed value to indicate which processor should be used. Suppose your messages contain a header indicating different levels. By using the sticky strategy, you can have messages with the same level choose the same processor over and over again.

In the Java DSL, you'd provide the expression by using a header expression, as shown here:

```
from("direct:start")
    .loadBalance().sticky(header("type"))
        .to("seda:a").to("seda:b")
    .end();
```

In XML, you'd do the following:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <sticky>
            <correlationExpression>
                <header>type</header>
            </correlationExpression>
        </sticky>
        <to uri="seda:a"/>
        <to uri="seda:b"/>
    </loadBalance>
</route>
```

The book's source code contains examples of using the strategies listed in table 5.9 in the chapter5/loadbalancer directory. To try the random, sticky, circuit breaker, or topic strategies, use the following Maven goals:

```
mvn test -Dtest=RandomLoadBalancerTest
mvn test -Dtest=SpringRandomLoadBalancerTest
mvn test -Dtest=StickyLoadBalancerTest
mvn test -Dtest=SpringStickyLoadBalancerTest
mvn test -Dtest=CircuitBreakerLoadBalancerTest
mvn test -Dtest=SpringCircuitBreakerLoadBalancerTest
mvn test -Dtest=TopicLoadBalancerTest
mvn test -Dtest=SpringTopicLoadBalancerTest
```

The failover strategy is a more elaborate strategy, which we cover

next.

5.6.3 USING THE FAILOVER LOAD BALANCER

Load balancing is often used to implement *failover*—the continuation of a service after a failure. The Camel failover load balancer detects the failure when an exception occurs and reacts by letting the next processor take over processing the message.

Given the following route snippet, the failover will always start by sending the messages to the first processor (`direct:a`) and only in the case of a failure will it let the next processor (`direct:b`) take over:

```
from("direct:start")
    .loadBalance().failover()
        .to("direct:a").to("direct:b")
    .end();
```

The equivalent snippet in XML is as follows:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <failover/>
        <to uri="direct:a"/>
        <to uri="direct:b"/>
    </loadBalance>
</route>
```

The book's source code contains this example in the `chapter5/loadbalancer` directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=FailoverLoadBalancerTest
mvn test -Dtest=SpringFailoverLoadBalancerTest
```

If you run the example, it will send in four messages. The second message will fail over and be processed by the `direct:b` processor. The other three messages will be processed successfully by `direct:a`.

In this example, the failover load balancer will react to any kind of exception being thrown, but you can provide it with a

number of exceptions to react to.

Suppose you want to fail over only if an `IOException` is thrown (which indicates communication errors with remote services, such as no connection). This is easy to configure, as shown in the Java DSL:

```
from("direct:start")
    .loadBalance().failover(IOException.class)
        .to("direct:a").to("direct:b")
    .end();
```

Here it's configured in XML:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <failover>
            <exception>java.io.IOException</exception>
        </failover>
        <to uri="direct:a"/>
        <to uri="direct:b"/>
    </loadBalance>
</route>
```

In this example, only one exception is specified, but you can specify multiple exceptions, as follows:

```
from("direct:start")
    .loadBalance().failover(IOException.class,
SQLException.class)
    .to("direct:a").to("direct:b")
    .end();
```

In XML, you do as follows:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <failover>
            <exception>java.io.IOException</exception>
            <exception>java.sql.SQLException</exception>
        </failover>
        <to uri="direct:a"/>
        <to uri="direct:b"/>
    </loadBalance>
```

```
</route>
```

You may have noticed in the failover examples that it always chooses the first processor and sends the failover to subsequent processors. You can think of this as the first processor being the master and the others slaves. But the failover load balancer also offers a strategy that combines round-robin with failure support.

USING FAILOVER WITH ROUND-ROBIN

The Camel failover load balancer in round-robin mode gives you the best of both worlds: it distributes the load evenly between the services, and it provides automatic failover.

In this scenario, you have three configuration options on the load balancer to dictate how it operates, as listed in table [5.10](#).

Table 5.10 Failover load-balancer configuration options

Config uration option	D ef a ul t	Description
maximu mFailo verAtt empts	- 1	<p>Specifies the number of failover attempts to try before exhausting (giving up):</p> <ul style="list-style-type: none">• Use -1 to attempt forever (never give up).• Use 0 to never fail over (give up immediately).• Use a positive value to specify a number of attempts. For example, a value of 3 will try up to three failover attempts before giving up.
inheri tError Handler	t r u e	Specifies whether Camel error handling is being used. When enabled, the load balancer will let the error handler be involved. If disabled, the load balancer will fail over immediately if an exception is thrown.

roundRobin	f a l s e	Specifies whether the load balancer operates in round-robin mode.
------------	-----------------------	---

To better understand the options in table 5.10 and how the round-robin mode works, let's start with a fairly simple example. In the Java DSL, you have to configure failover with all the options in bold:

```
from("direct:start")
    .loadBalance().failover(1, false, true)
        .to("direct:a").to("direct:b")
    .end();
```

In this example, the `maximumFailoverAttempts` option is set to `1`, which means it will at most try to fail over once (it will make one attempt for the initial request and one more for the failover attempt). If both attempts fail, Camel will propagate the exception back to the caller. The second parameter is set to `false`, which means it isn't inheriting Camel's error handling. This allows the failover load balancer to fail over immediately when an exception occurs, instead of having to wait for the Camel error handler to give up first. The last parameter indicates that it's using the round-robin mode.

In XML, you configure the options as attributes on the `failover` tag:

```
<route>
    <from uri="direct:start"/>
    <loadBalance inheritErrorHandler="false">
        <failover roundRobin="true"
maximumFailoverAttempts="1"

```

The book's source code contains this example in the `chapter5/loadbalancer` directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=FailoverRoundRobinLoadBalancerTest  
mvn test -Dtest=SpringFailoverRoundRobinLoadBalancerTest
```

If you're curious about the `inheritErrorHandler` configuration option, take a look at the following examples in the book's source code:

```
mvn test -Dtest=FailoverInheritErrorHandlerLoadBalancerTest  
mvn test -  
Dtest=SpringFailoverInheritErrorHandlerLoadBalancerTest
```

This concludes our tour of the failover load balancer. The next section explains how to implement and use your own custom strategy, which you may want to do when you need to use special load-balancing logic.

5.6.4 USING A CUSTOM LOAD BALANCER

Custom load balancers allow you to be in full control of the balancing strategy in use. For example, you could build a strategy that acquires load statistics from various services and picks the service with the lowest load.

Let's look at an example. Suppose you want to implement a priority-based strategy that sends gold messages to a certain processor and the remainder to a secondary destination.

Figure 5.13 illustrates this principle.

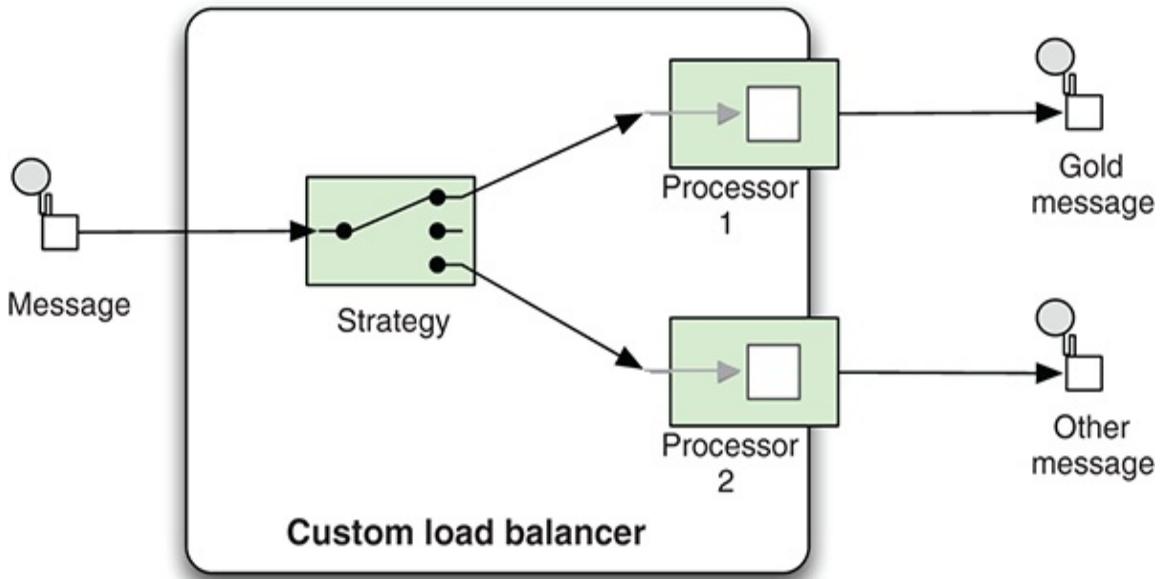


Figure 5.13 Using a custom load balancer to route gold messages to processor 1 and other messages to processor 2

When implementing a custom load balancer, you'll often extend the `SimpleLoadBalancerSupport` class, which provides a good starting point. The following listing shows how to implement a custom load balancer.

Listing 5.11 Custom load balancer

```

public class MyCustomLoadBalancer extends
SimpleLoadBalancerSupport {
    public boolean process(Exchange exchange) throws
Exception {
        Processor target = chooseProcessor(exchange);
        target.process(exchange);
    }
    @Override
    protected Processor chooseProcessor(Exchange exchange)
{
    String type = exchange.getIn().getHeader("type",
String.class);
    if ("gold".equals(type)) {
        return getProcessors().get(0);
    }
}

```

①

selects processor 1

```
    } else {
        return getProcessors().get(1);
```

2

selects processor 2

```
    }
}
```

As you can see, it doesn't take much code. In the process method, you invoke the chooseProcessor method, which is the strategy that picks the processor to process the message. In this example, it'll pick the first processor if the message is a gold type, and the second processor if not.

In the Java DSL, you use a custom load balancer as shown in bold:

```
from("direct:start")
    .loadBalance(new MyCustomLoadBalancer())
        .to("seda:a").to("seda:b")
    .end();
```

In XML, you need to declare a <bean> tag:

```
<bean id="myCustom"
class="camelinaction.MyCustomLoadBalancer"/>
```

You then refer to that bean from the <custom> tag inside the <loadBalance> tag:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <custom ref="myCustom"/>
        <to uri="seda:a"/>
        <to uri="seda:b"/>
    </loadBalance>
</route>
```

The book's source code contains this example in the chapter5/loadbalancer directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=CustomLoadBalancerTest  
mvn test -Dtest=SpringCustomLoadBalancerTest
```

3

We've now covered the Load Balancer EIP in Camel, which brings us to the end of our long journey to visit five great EIPs implemented in Camel.

5.7 Summary and best practices

Since the arrival of the *Enterprise Integration Patterns* book on the scene, we've had a common vocabulary, graphical notation, and concepts for designing applications to tackle today's integration challenges. You've encountered these EIPs throughout this book. In chapter 2 we reviewed the most common patterns, and this chapter reviews five of the most complex and sophisticated patterns in great detail. You may view the EIP book as the theory, and Camel as the software implementation of the book.

Here are some EIP best practices to take away from this chapter:

- *Learn the patterns*—Take the time to study the EIPs, especially the common patterns covered in chapter 2 and those presented in this chapter. Consider getting the EIP book to read more about the patterns; great advice is given in the book. The EIP book authors also maintain a website, which is a nice reference for the patterns. Lately they've even updated it to include concrete examples using Camel:
www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html. The patterns are universal, and the knowledge you gain when using EIPs with Camel is something you can take with you.
- *Use the patterns*—If you have a problem you don't know how to resolve, there's a good chance others have scratched that itch before. Consult the EIP book and the online Camel patterns catalog: <http://camel.apache.org/eip>. Another highly

recommended EIP pattern book is *Camel Design Patterns* by Bilgin Ibryam (Leanpub, 2016, <https://leanpub.com/camel-design-patterns>).

- *Start simply*—When learning to use an EIP, you should create a simple test to try the pattern and learn how to use it. Having too many new moving parts in a Camel route can clutter your view and make it difficult to understand what's happening and, maybe, why it doesn't do what you expect.
- *Come back to this chapter*—If you're going to use any of the five EIPs covered in this chapter, we recommend you reread the relevant parts of the chapter. These patterns are sophisticated and have many features and options to tweak.

The next chapter covers the use of components with Camel. You've already used components, such as the file and SEDA components. But there's much more to components, so we devote an entire chapter to cover them in detail.

6

Using components

This chapter covers

- An overview of Camel components
- Working with files and databases
- Messaging with JMS
- Networking with Netty
- Working with databases
- In-memory messaging
- Automating tasks with the Quartz and Scheduler components
- Sending and receiving email

So far, we've touched on only a handful of ways that Camel can communicate with external applications, and we haven't gone into much detail on most components. It's time to take your use of the components you've already seen to the next level, and to introduce new components that will enable your Camel applications to communicate with the outside world.

First, we'll discuss exactly what it means to be a component in Camel. We'll also show you how components are added to Camel. Then, although we can't describe every component—that would at least triple the length of this book—we'll present the

most commonly used ones.

Table 6.1 lists the components covered in this chapter and the URLs for their official documentation.

Table 6.1 Components covered in this chapter

Component function	Component	Camel documentation reference
File I/O	File	http://camel.apache.org/file2.html
	FTP	http://camel.apache.org/ftp2.html
Asynchronous messaging	JMS	http://camel.apache.org/jms.html
Networking	Netty4	http://camel.apache.org/netty4.html
Working with databases	JDBC	http://camel.apache.org/jdbc.html
	JPA	http://camel.apache.org/jpa.html
In-memory messaging	Direct	http://camel.apache.org/direct.html
	Direct-VM	http://camel.apache.org/direct-vm.html
	SEDA	http://camel.apache.org/seda.html
	VM	http://camel.apache.org/vm.html
Automating tasks	Scheduler	http://camel.apache.org/scheduler.html
	Quartz2	http://camel.apache.org/quartz2.html

Let's start with an overview of Camel components.

6.1 Overview of Camel components

Components are the primary extension point in Camel. Over the years since Camel's inception, the list of components has grown. As of version 2.20.1, Camel ships with more than 280 components, and dozens more are available separately from other community sites.¹ These components allow you to bridge to many APIs, protocols, data formats, and so on. Camel saves

you from having to code these integrations yourself; thus it achieves its primary goal of making integration easier.

¹ See appendix B for information on some of these community sites.

What does a Camel component look like? Well, if you think of Camel routes as highways, components are roughly analogous to on- and off-ramps. A message that travels down a route needs to take an off-ramp to get to another route or external service. If the message is headed for another route, it then needs to take an on-ramp to get onto that route.

From an API point of view, a Camel component is simple, consisting of a class implementing the `Component` interface, shown here:

```
public interface Component extends CamelContextAware {  
    Endpoint createEndpoint(String uri) throws Exception;  
    boolean useRawUri();  
}
```

The main responsibility of a component is to be a factory for endpoints. To do this, a component also needs to extend `CamelContextAware`, which means it holds a reference to `CamelContext`. `CamelContext` provides access to Camel's common facilities, such as the registry, class loader, and type converters. This relationship is shown in [figure 6.1](#).



Figure 6.1 A component creates endpoints and may use the `CamelContext`'s facilities to accomplish this.

Components are added to a Camel runtime in two main ways: by manually adding them to `CamelContext` and through autodiscovery.

6.1.1 MANUALLY ADDING COMPONENTS

You've seen the manual addition of a component already. In

chapter 2, you had to add a configured JMS component to CamelContext to use ConnectionFactory. This was done using the addComponent method of the camelContext interface, as follows:

```
CamelContext context = new DefaultCamelContext();
context.addComponent("jms",
    JmsComponent.jmsComponentAutoAcknowledge(connectionFactory)
);
```

In this example, you add a component created by the JmsComponent.jmsComponent-AutoAcknowledge method and assign it a name of jms. This component can be selected in a URI by using the jms scheme.

6.1.2 AUTODISCOVERING COMPONENTS

The other way components can be added to Camel is through autodiscovery. The autodiscovery process is illustrated in figure 6.2.

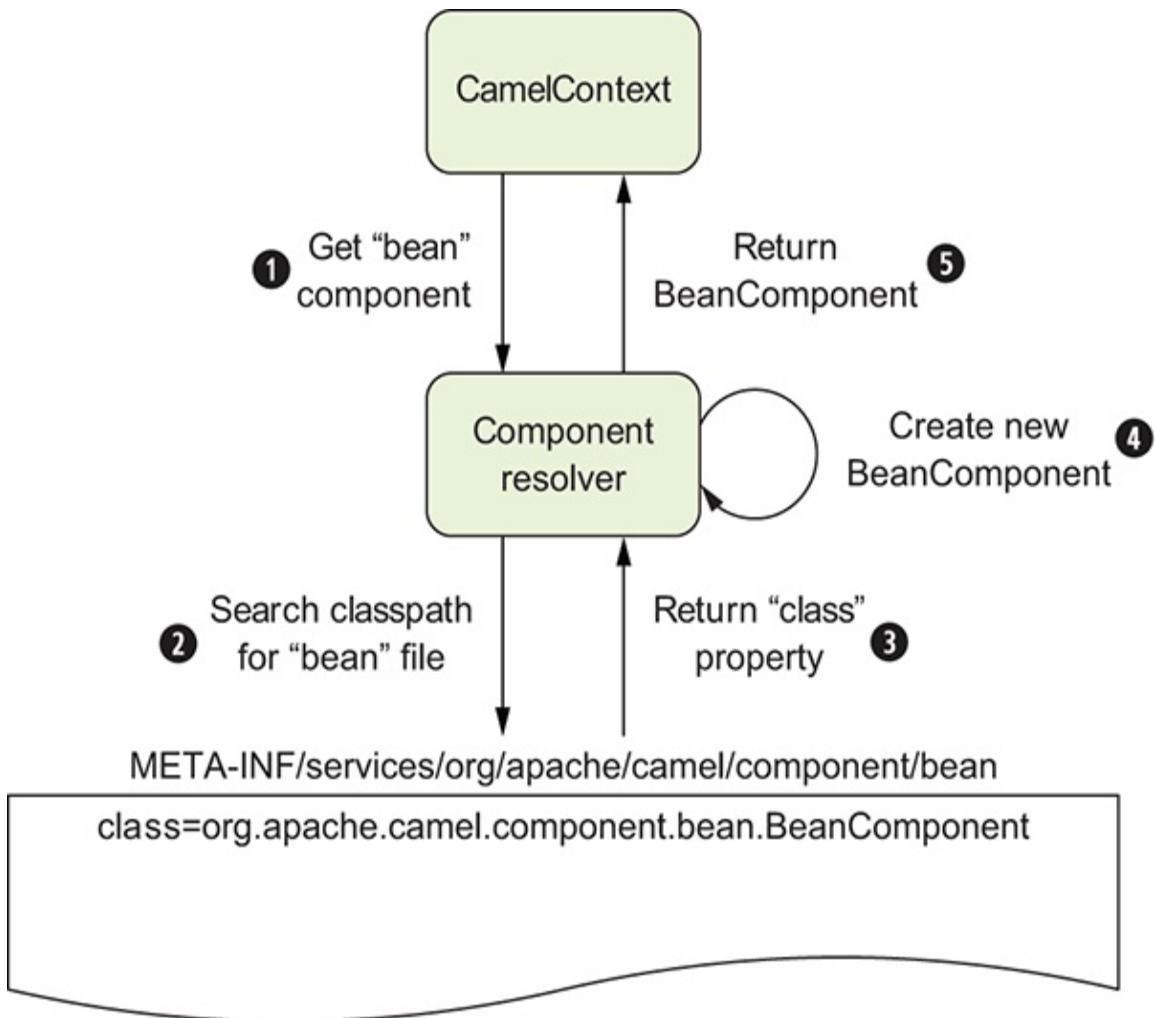


Figure 6.2 To autodiscover a component named “bean,” the component resolver searches for a file named “bean” in a specific directory on the classpath. This file specifies that the component class that will be created is BeanComponent.

Autodiscovery is the way the components that ship with Camel are registered. To discover new components, Camel looks in the META-INF/services/org/apache/camel/component directory on the classpath for files. Files in this directory determine the name of a component and the fully qualified class name.

As an example, let’s look at the Bean component. It has a file named bean in the META-INF/services/org/apache/camel/component directory that contains a single line:

```
class=org.apache.camel.component.bean.BeanComponent
```

This class property tells Camel to load up the `org.apache.camel.component.bean.BeanComponent` class as a new component, and the filename gives the component the name of Bean.

TIP We discuss how to create your own Camel component in section 8.4 in chapter 8.

Most of the components in Camel are in separate Maven modules from the camel-core module, because they usually depend on third-party dependencies that would bloat the core. For example, the Atom component depends on Apache Abdera to communicate over Atom. You wouldn't want to make every Camel application depend on Abdera, so the Atom component is included in a separate camel-atom module.

The camel-core module has 24 useful components built in, though. These are listed in [table 6.2](#).

Table 6.2 Components in the camel-core module

Component	Description	Camel documentation reference
Bean	Invokes a Java bean in the registry. You saw this used extensively in chapter 4.	http://camel.apache.org/bean.html
Browse	Allows you to browse the list of exchanges that passed through a browse endpoint. This can be useful for testing, visualization, or debugging.	http://camel.apache.org/browse.html
Class	Creates a new Java bean based on a class name and invokes the bean similar to the bean component.	http://camel.apache.org/class.html
C	Allows you to control the lifecycle of routes and	http://camel.apache.org/co

o n t r ol B us	gather performance statistics by sending messages to a ControlBus endpoint. Based on the Control Bus EIP pattern.	ntrolbus-component.html
D at a F o r m at	A convenience component that allows you to invoke a data format as a component.	http://camel.apache.org/da taformat-component.html
D at a S et	Allows you to create large numbers of messages for soak or load testing.	http://camel.apache.org/da taset.html
D ir e ct	Allows you to synchronously call another endpoint with little overhead. Section 6.6 covers this component.	http://camel.apache.org/direct.html
D ir e ct - VM	Allows you to synchronously call another endpoint in the same JVM. Section 6.6 covers this component.	http://camel.apache.org/direct-vm.html
Fi le	Reads or writes to files. Section 6.2 covers this component.	http://camel.apache.org/file 2.html
L a n g u a ge	Executes a script against the incoming exchange by using one of the languages supported by Camel.	http://camel.apache.org/lan guage.html
Log	Logs messages to various logging providers.	http://camel.apache.org/log .html

M o ck	Tests that messages flow through a route as expected. You'll see the Mock component in action in chapter 9.	http://camel.apache.org/mock.html
P r o p e r t i e s	Allows you to use property placeholders in endpoint URIs. You've seen this already in section 2.5.2.	http://camel.apache.org/properties.html
R ef	Looks up endpoints in the registry.	http://camel.apache.org/ref.html
R E ST	Used for hosting or calling REST services. See section 10.2 for more information on how this is used with Camel's Rest DSL.	http://camel.apache.org/rest.html
R E S T A PI	Used to provide Swagger API docs for REST endpoints created with Camel's Rest DSL. Covered in section 10.3.	https://github.com/apache/camel/blob/master/camel-core/src/main/docs/rest-api-component.adoc
S E DA	Allows you to asynchronously call another endpoint in the same <code>CamelContext</code> . Section 6.6 covers this component.	http://camel.apache.org/seda.html
S c h e d ul er	Sends out messages at regular intervals. You'll learn more about the Scheduler component and a more powerful scheduling endpoint based on Quartz in section 6.7.	http://camel.apache.org/scheduler.html
S tub	Allows you to stub out real endpoint URIs for development or testing purposes.	http://camel.apache.org/stub.html
T e st	Tests that messages flowing through a route match expected messages pulled from another endpoint.	http://camel.apache.org/test.html
Ti m er	Sends out messages at regular intervals.	http://camel.apache.org/timer.html

V al id at or	Validates the message body by using the JAXP Validation API.	http://camel.apache.org/validation.html
VM	Allows you to asynchronously call another endpoint in the same JVM. Section 6.6 covers this component.	http://camel.apache.org/vm.html
X S LT	Transforms a message by using an XSLT template.	http://camel.apache.org/xslt.html

Now let's look at each component from table 6.1 in detail. We'll start with the File component.

6.2 Working with files: File and FTP components

It seems that in integration projects, you always end up needing to interface with a filesystem somewhere. You may find this strange, as new systems often provide nice web services and other remoting APIs to serve as integration points. The problem is that in integration, you often have to deal with older legacy systems, and file-based integrations are common.

For example, you might need to read a file that was written by another application—it could be sending a command to be executed, an order to be processed, data to be logged, or anything else. This kind of information exchange, illustrated in figure 6.3, is called a *file transfer* in EIP terms.

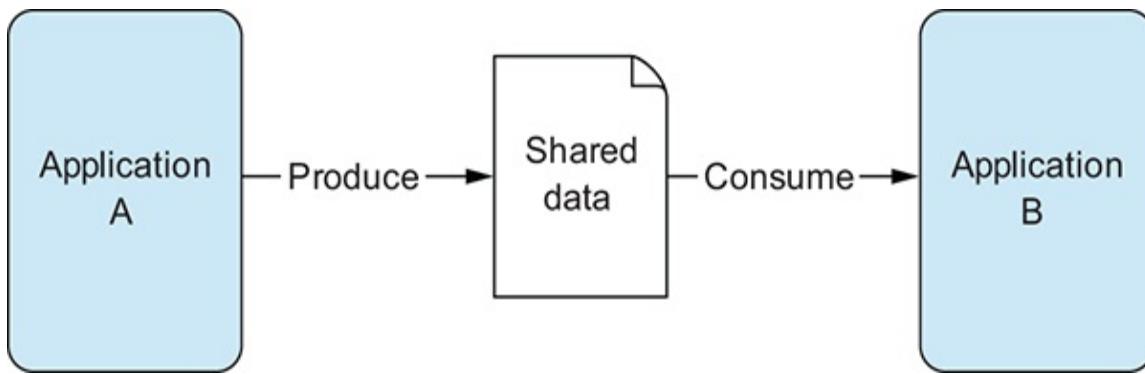


Figure 6.3 A file transfer between two applications is a common way to integrate with legacy systems.

Another reason that file-based integrations are so common is that they're easy to understand. Even novice computer users know something about filesystems.

Even though they're easy to understand, file-based integrations are difficult to get right. Developers commonly have to battle with complex I/O APIs, platform-specific filesystem issues, different file encodings, concurrent access, and the like.

Camel has extensive support for interacting with filesystems. This section covers how to use the File component to read files from and write them to the local filesystem. This section also covers advanced options for file processing and ways to access remote files with the FTP component.

6.2.1 READING AND WRITING FILES WITH THE FILE COMPONENT

As you saw before, the File component is configured through URI options. **Table 6.3** shows some common options; for a complete listing, see the online documentation (<http://camel.apache.org/file2.html>).

Table 6.3 Common URI options used to configure the File component

O	D	Description
pt	ef	
io	a	
n	ul	
	t	
	v	

	al u e	
d e l a y	5 0 0	Specifies the number of milliseconds between polls of the directory.
r e c u r s i v e	f a l s e	Specifies whether to recursively process files in all subdirectories of this directory.
n o o p	f a l s e	Specifies file-moving behavior. By default, Camel moves files to the .camel directory after processing them. To stop this behavior and keep the original files in place, set the noop option to true.
f i l e N a m e		Uses an expression to set the filename used. For consumers, this acts as a filename filter; in producers, it's used to set the name of the file being written.
f i l e E x i s t	o v e r r i d e	Specifies what a file producer will do if the same filename already exists. Valid options are <code>Override</code> , <code>Append</code> , <code>Fail</code> , <code>Ignore</code> , <code>Move</code> , and <code>TryRename</code> . <code>Override</code> causes the file to be replaced. <code>Append</code> adds content to the file. <code>Fail</code> causes an exception to be thrown. If <code>Ignore</code> is set, an exception won't be thrown, and the file won't be written. <code>Move</code> causes the existing file to be moved. <code>Move</code> also requires the <code>moveExisting</code> option to be set to an expression used to compute the filename to move to. <code>TryRename</code> causes the file rename to be attempted. <code>TryRename</code> applies only to temporary files specified by the <code>tempFileName</code> option.
d e l e t e	f a l s e	Specifies whether Camel will delete the file after processing. By default, Camel won't delete the file.
m o	.	Specifies the directory to which Camel moves files after it's done processing them

		processing item.
v e	a m e 1	
i n c l u d e		Specifies a regular expression. Camel processes only those files that match this expression. For example, <code>include=.*xml\$</code> would include all files with the .xml extension.
e x c l u d e		Specifies a regular expression. Camel excludes files based on this expression.

Let's first see how Camel can be used to read files.

READING FILES

As you've seen in previous chapters, reading files with Camel is straightforward. Here's a simple example:

```
public void configure() {
    from("file:data/inbox?noop=true").to("stream:out");
}
```

This route reads files from the data/inbox directory and prints the contents of each to the console. The printing is done by sending the message to the `System.out` stream, accessible by using the Stream component. As stated in table 6.3, the `noop` flag tells Camel to leave the original files as is. This is a convenience option for testing, because it means that you can run the route many times without having to repopulate a directory of test files.

To run this yourself, change to the chapter6/file directory in the book's source code and run this command:

```
mvn test -Dtest=FilePrinterTest
```

What if you remove the `noop` flag and change the route to the

following?

```
public void configure() {  
    from("file:data/inbox").to("stream:out");  
}
```

This uses Camel's default behavior, which is to move the consumed files to a special camel directory (though the directory can be changed with the `move` option); the files are moved after the routing has completed. This behavior was designed so that files wouldn't be processed over and over, but it also keeps the original files around in case something goes wrong. If you don't mind losing the original files, you can use the `delete` option listed in table [6.3](#).

By default, Camel also locks any files that are being processed. The locks are released after routing is complete.

Both of the two preceding routes consume any file not beginning with a period, so they ignore files such as `.camel`, `.m2`, and so on. You can customize which files are included by using the `include` and `exclude` options.

WRITING FILES

You just saw how to read files created by other applications or users. Now let's see how Camel can be used to write files. Here's a simple example:

```
<route>  
    <from uri="stream:in?promptMessage=Enter something:"/>  
    <to uri="file:data/outbox"/>  
</route>
```

This example uses the Stream component to accept input from the console. The `stream:in` URI instructs Camel to read any input from `System.in` on the console and create a message from that. The `promptMessage` option displays a prompt, so you know when to enter text. The `file:data/outbox` URI instructs Camel to write out the message body to the `data/outbox` directory.

To see what happens firsthand, you can try the example by

changing to the chapter6/file directory in the book's source code and executing the following command:

```
mvn camel:run
```

When this runs, you'll see an `Enter something:` prompt. Enter text into the console and press Enter, like this:

```
Enter something:Hello
```

The example keeps running until you press Ctrl-C. The text (in this case, `Hello`) is read in by the Stream component and added as the body of a new message. This message's body (the text you entered) is then written out to a file in the data/outbox directory (which will be created if it doesn't exist).

If you run a directory listing on the data/outbox directory now, you'll see a single file that has a rather strange name:

```
ID-ghost-43901-1489018386363-0-1
```

Because you didn't specify a filename to use, Camel chose a unique filename based on the message ID.

To set the filename that should be used, you can add a `fileName` option to your URI. For example, you could change the route so it looks like this:

```
<route>
  <from uri="stream:in?promptMessage=Enter something:"/>
  <to uri="file:data/outbox?fileName=prompt.txt"/>
</route>
```

Now, any text entered into the console will be saved into the `prompt.txt` file in the data/outbox directory.

Camel will by default overwrite `prompt.txt`, so you now have a problem with this route. If text is frequently entered into the console, you may want new files created each time, so they don't overwrite the old ones. To implement this in Camel, you can use an expression for the filename. You can use the Simple expression language to put the current time and date information into your filename:

```
<route>
  <from uri="stream:in?promptMessage=Enter something:"/>
  <to uri="file:data/outbox?fileName=${date:now:yyyyMMdd-
hh:mm:ss}.txt"/>
</route>
```

The `date:now` expression returns the current date, and you can also use any formatting options permitted by `java.text.SimpleDateFormat`.

Now if you enter text into the console at 2:00 p.m. on January 10, 2009, the file in the `data/outbox` directory will be named something like this:

```
20090110-02:00:53.txt
```

The simple techniques for reading from and writing to files discussed here will be adequate for most of the cases you'll encounter in the real world. For the trickier cases, many configuration possibilities are listed in the online documentation.

We've started slowly with the File component, to get you comfortable with using components in Camel. Next we present the FTP component, which builds on the File component but introduces messaging across a network. After that, we'll get into more complex topics.

6.2.2 ACCESSING REMOTE FILES WITH THE FTP COMPONENT

Probably the most common way to access remote files is by using FTP, and Camel supports three flavors of FTP:

- Plain FTP mode transfer
- Secure FTP (SFTP) for secure transfer
- FTP Secure (FTPS) for transfer with the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) cryptographic protocols enabled

The FTP component inherits all the features and options of the

File component, and it adds a few more options, as shown in table 6.4. For a complete listing of options for the FTP component, see the online documentation (<http://camel.apache.org/ftp2.html>).

Table 6.4 Common URI options used to configure the FTP component

Option	Default value	Description
username		Provides a username to the remote host for authentication. If no username is provided, anonymous login is attempted. You can also specify the username by prefixing <code>username@</code> to the hostname in the URI.
password		Provides a password to the remote host to authenticate the user. You can also specify the password by prefixing the hostname in the URI with <code>username:password@</code> .
binary	false	Specifies the transfer mode. By default, Camel transfers in ASCII mode; set this option to <code>true</code> to enable binary transfer.
disconnect	false	Specifies whether Camel will disconnect from the remote host right after use. The default is to remain connected.
maximumReconnectAttempts	3	Specifies the maximum number of attempts Camel will make to connect to the remote host. If all these attempts are unsuccessful, Camel will throw an exception. A value of 0 disables this feature.
reconnectDelay	1000	Specifies the delay in milliseconds between reconnection attempts.

Because the FTP component isn't part of the camel-core module, you need to add a dependency to your project. If you use Maven, you add the following dependency to your POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>2.20.1</version>
</dependency>
```

To demonstrate accessing remotes files, let's use the Stream component as in the previous section to interactively generate and send files over FTP. A route that accepts text on the console and then sends it over FTP would look like this:

```
<route>
  <from uri="stream:in?promptMessage=Enter something:" />
  <to
uri="ftp://rider:secret@localhost:21000/target/data/outbox"
/>
</route>
```

This is a Spring-based route; Spring makes it easy to hook start and stop methods to an embedded FTP server. This FTP endpoint URI specifies that Camel should send the message to an FTP server on the localhost listening on port 21000, using rider as the username and secret as the password. It also specifies that messages are to be stored in the data/outbox directory of the FTP server.

To run this example, change to the chapter6/ftp directory and run this command:

```
mvn camel:run
```

After Camel has started, you need to enter something into the console:

```
Enter something:Hello
```

The example keeps running until you press Ctrl-C.

You can now check to see whether the message made it into the FTP server. The FTP server's root directory was set up to be the current directory of the application, so you can check data/outbox for a message:

```
$ cat target/data/outbox/ID-ghost-43901-1489018386363-0-1
Hello
```

As you can see, using the FTP component is similar to using the File component.

Now that you know how to do the most basic of integrations with files and FTP, let's move on to more advanced topics, such as JMS and web services.

6.3 Asynchronous messaging: JMS component

JMS messaging is an incredibly useful integration technology. It promotes loose coupling in application design, has built-in support for reliable messaging, and is by nature asynchronous. As you saw in chapter 2, when you looked at JMS, it's also easy to use from Camel. This section expands on the coverage in chapter 2 by going over some of the more commonly used configurations of the JMS component.

Camel doesn't ship with a JMS provider; you need to configure Camel to use a specific JMS provider by passing in a ConnectionFactory instance. For example, to connect to an Apache ActiveMQ broker listening on port 61616 of the local host, you could configure the JMS component like this:

```
<bean id="jms"
  class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean
      class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL"
        value="tcp://localhost:61616"/>
    </bean>
  </property>
</bean>
```

The `tcp://localhost:61616` URI passed in to `ConnectionFactory` is JMS provider-specific. In this example, you're using the `ActiveMQConnectionFactory`, so the URI is parsed by ActiveMQ. The URI tells ActiveMQ to connect to a broker by using TCP on port 61616 of the local host.

If you want to connect to a broker over another protocol, ActiveMQ supports connections over VM, SSL, UDP, multicast,

MQTT, AMQP, and so on. Throughout this section, we'll demonstrate JMS concepts using ActiveMQ as the JMS provider, but any provider could be used here.

The ActiveMQ component

By default, a JMS ConnectionFactory doesn't pool connections to the broker, so it spins up new connections for every message. The way to avoid this is to use connection factories that use connection pooling.

For convenience to Camel users, ActiveMQ ships with the ActiveMQ component, which automatically configures connection pooling for improved performance. The ActiveMQ component is used as follows:

```
<bean id="activemq"
      class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL"
      value="tcp://localhost:61616"/>
  </bean>
```

When using this component, you also need to depend on the activemq-camel module from ActiveMQ:

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-camel</artifactId>
  <version>5.15.2</version>
</dependency>
```

This module contains the Camel ActiveMQ component.

Camel's JMS component has a daunting list of configuration options—more than 80 to date. Many of these are seen in only specific JMS usage scenarios. The common ones are listed in table 6.5.

TIP Apache Kafka is another popular asynchronous messaging project. We cover this in section 17.4 of chapter 17.

To use the JMS component in your project, you need to include the camel-jms module on your classpath as well as any JMS provider JARs. If you’re using Maven, the JMS component can be added with the following dependency:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jms</artifactId>
  <version>2.20.1</version>
</dependency>
```

Table 6.5 Common URI options used to configure the JMS component

Option	Default value	Description
clientID		Sets the JMS client ID, which must be unique among all connections to the JMS broker. The client ID set in ConnectionFactory overrides this one if set. Needed only when using durable topics.
concurrentConsumers	1	Sets the number of consumer threads to use. It’s a good idea to increase this for high-volume queues, but it’s not advisable to use more than one concurrent consumer for JMS topics, because this will result in multiple copies of the same message.
disableReplyTo	false	Specifies whether Camel should ignore the <code>JMSReplyTo</code> header in any messages. Set this if you don’t want Camel to send a reply back to the destination specified in the <code>JMSReplyTo</code> header.
durableSubscriptionName		Specifies the name of the durable topic subscription. The <code>clientId</code> option must also be set.

ame		
maxC oncu rren tCon sume rs	1	Sets the maximum number of consumer threads to use. If this value is higher than concurrentConsumers, new consumers are started dynamically as load demands. If load drops, these extra consumers will be freed and the number of consumers will be equal to concurrentConsumers again. Increasing this value isn't advisable when using topics.
rep1 yTo		Sets the destination that the reply is sent to. This overrides the JMSReplyTo header in the message. By setting this, Camel will use a fixed reply queue. By default, Camel uses a temporary reply queue.
rep1 yToC oncu rren tCon sume rs	1	Sets the number of threads to use for request-reply style messaging. This value is separate from concurrentConsumers.
rep1 yToM axCo ncur rent Cons umer s	1	Sets the maximum number of threads to use for request-reply style messaging. This value is separate from maxConcurrentConsumers.
requ estT imeo ut	2 0 0 0 0	Specifies the time in milliseconds before Camel will time out when sending a message in request-reply mode. You can override the endpoint value by setting the CamelJmsRequestTimeout header on a message.
sele ctor		Sets the JMS message selector expression. Only messages passing this predicate will be consumed.
tran sact ed	f a l s e	Enables transacted sending and receiving of messages in Inonly mode.

The SJMS component

The Camel JMS component is built on top of the Spring

JMS library, so many of the options map directly to a Spring org.springframework.jms.listener.AbstractMessageListenerContainer used under the hood. Furthermore, the dependency on Spring brings in a whole set of other Spring JARs. For memory-sensitive deployments, or those where you aren't using Spring at all, Camel provides the SJMS component. The S in SJMS stands for *Simple* and *Spring-less*.

To use this component, you need to add camel-sjms2 to your Maven pom.xml:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sjms2</artifactId>
  <version>2.20.1</version>
</dependency>
```

The best way to show that Camel is a great tool for JMS messaging is with an example. Let's look at how to send and receive messages over JMS.

6.3.1 SENDING AND RECEIVING MESSAGES

In chapter 2, you saw how orders are processed at Rider Auto Parts. The process started out as a step-by-step procedure: orders were first sent to accounting to validate the customer standing and then to production for manufacture. This process was improved by sending orders to accounting and production at the same time, cutting out the delay involved when production waited for the okay from accounting. A multicast EIP was used to implement this scenario.

Figure 6.4 illustrates another possible solution: using a JMS topic following a publish-subscribe model. In that model, listeners such as accounting and production can subscribe to the topic, and new orders are published to the topic. In this way, both accounting and production receive a copy of the order message.

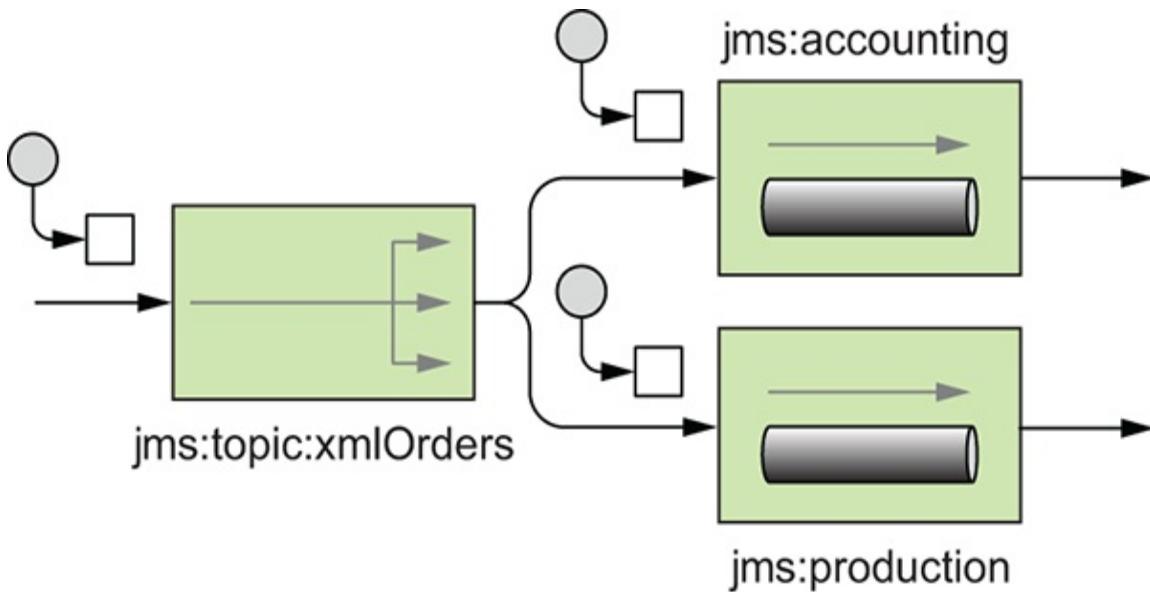


Figure 6.4 Orders are published to the `xmlOrders` topic, and the two subscribers (the accounting and production queues) get a copy of the order.

To implement this in Camel, you set up two consumers, which means two routes are needed:

```
from("jms:topic:xmlOrders").to("jms:accounting");
from("jms:topic:xmlOrders").to("jms:production");
```

When a message is sent (published) to the `xmlOrders` topic, both the accounting and production queues receive a copy.

As you saw in chapter 2, an incoming order could originate from another route (or set of routes), such as one that receives orders via a file, as shown in the following listing.

Listing 6.1 Topics allow multiple receivers to get a copy of the message

```
from("file:src/data?noop=true").to("jms:incomingOrders");
from("jms:incomingOrders")
    .choice()
        .when(header("CamelFileName").endsWith(".xml"))
            .to("jms:topic:xmlOrders") ①
```

XML orders are routed to `xmlOrders` topic

```
.when(header("CamelFileName").regex("^.*  
(csv|csl)$"))  
    .to("jms:topic:csvOrders");  
from("jms:topic:xmlOrders").to("jms:accounting"); ②
```

Both listening queues get copies

```
from("jms:topic:xmlOrders").to("jms:production"); ②
```

To run this example, go to the chapter6/jms directory in the book's source and run this command:

```
mvn camel:run
```

This outputs the following on the command line:

```
Accounting received order: message1.xml  
Production received order: message1.xml
```

Why do you get this output? Well, you have a single order file named message1.xml, and it's published to the xmlOrders topic. Both the accounting and production queues are subscribed to the topic, so each receives a copy. Testing routes consume the messages on those queues and output the messages.

To send an order to the topic by using ProducerTemplate, you could use the following snippet:

```
ProducerTemplate template =  
camelContext.createProducerTemplate();  
template.sendBody("jms:topic:xmlOrders", "<?xml ...");
```

This is a useful feature for getting direct access to any endpoint in Camel.

All the JMS examples so far have been one-way only. Let's look at how to deliver a reply to the sent message.

6.3.2 REQUEST-REPLY MESSAGING

JMS messaging with Camel (and in general) is asynchronous by default. Messages are sent to a destination, and the client doesn't wait for a reply. But at times it's useful to be able to wait and get

a reply after sending to a destination. One obvious application is when the JMS destination is a front end to a service—in this case, a client sending to the destination would be expecting a reply from the service.

JMS supports this type of messaging by providing a `JMSReplyTo` header, so that the receiver knows where to send the reply, and a `JMSCorrelationID`, used to match replies to requests if multiple replies are awaiting. This flow of messages is illustrated in figure 6.5.

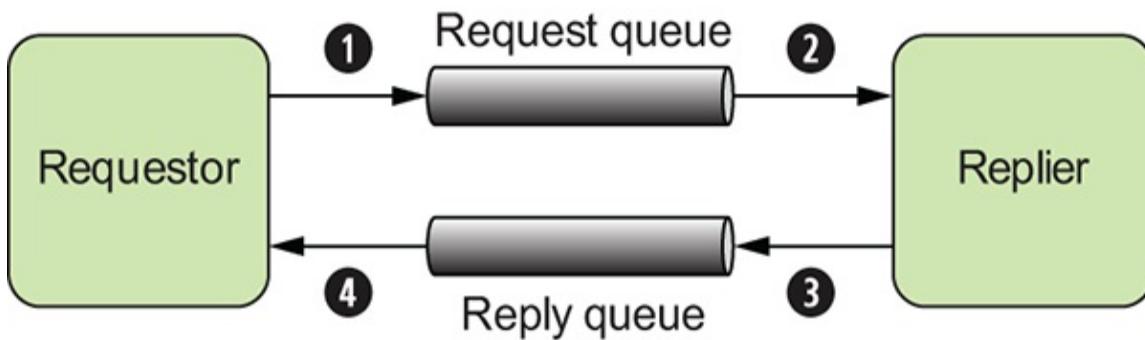


Figure 6.5 In request-reply messaging, a requestor sends a message to a request queue ① and then waits for a reply in the reply queue ④. The replier waits for a new message in the request queue ②, inspects the `JMSReplyTo` address, and then sends a reply back to that destination ③.

Camel takes care of this style of messaging so you don't have to create special reply queues, correlate reply messages, and the like. By changing the message exchange pattern (MEP) to `InOut`, Camel will enable request-reply mode for JMS.

To demonstrate, let's take a look at an order validation service within the Rider Auto Parts back-end systems that checks orders against the company database to make sure the parts listed are actual products. This service is exposed via a queue named `validate`. The route exposing this service over JMS could be as simple as this:

```
from("jms:validate").bean(ValidatorBean.class);
```

When calling this service, you need to tell Camel to use request-reply messaging by setting the MEP to `InOut`. You can use the

`exchangePattern` option to set this as follows:

```
from("jms:incomingOrders").to("jms:validate?  
exchangePattern=InOut")...
```

You can also specify the MEP by using the `inOut` DSL method:

```
from("jms:incomingOrders").inOut().to("jms:validate")...
```

With the `inOut` method, you can even pass in an endpoint URI as an argument, which shortens your route:

```
from("jms:incomingOrders").inOut("jms:validate")...
```

By specifying an `Inout` MEP, Camel will send the message to the validate queue and wait for a reply on a temporary queue that it creates automatically. When the `validatorBean` returns a result, that message is propagated back to the temporary reply queue, and the route continues on from there.

Rather than using temporary queues, you can explicitly specify a reply queue. You can do that by setting the `JMSReplyTo` header on the message or by using the `replyTo` URI option described in table [6.5](#).

A handy way of calling an endpoint that can return a response is by using the request methods of the `ProducerTemplate`. For example, you can send a message into the `incomingOrders` queue and get a response back with the following call:

```
Object result = template.requestBody("jms:incomingOrders",  
    "<order name=\"motor\" amount=\"1\""  
    "customer=\"honda\"/>");
```

This returns the result of the `validatorBean`.

To try this out, go to the `chapter6/jms` directory in the book's source and run this command:

```
mvn test -Dtest=RequestReplyJmsTest
```

The command runs a unit test demonstrating request-reply messaging as discussed in this section.

In the JMS examples you've looked at so far, several data mappings have been happening behind the scenes—mappings that are necessary to conform to the JMS specification. Camel could be transporting any type of data, so that data needs to be converted to a type that JMS supports. We'll look into this next.

6.3.3 MESSAGE MAPPINGS

Camel hides a lot of the details when doing JMS messaging, so you don't have to worry about them. But one detail you should be aware of is that Camel maps both bodies and headers from the arbitrary types and names allowed in Camel to JMS-specific types.

BODY MAPPING

Although Camel poses no restrictions on the content of a message body, JMS specifies different message types based on the body type. [Figure 6.6](#) shows the five concrete JMS message implementations.

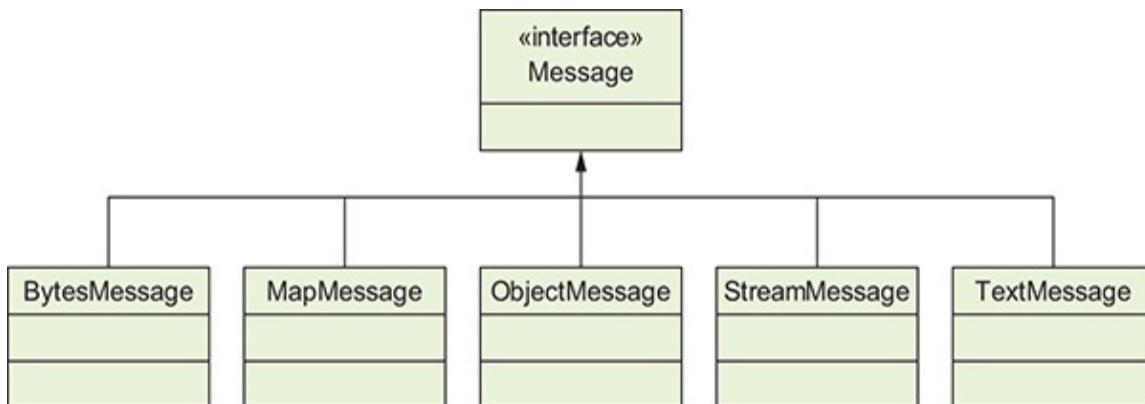


Figure 6.6 The `javax.jms.Message` interface has five implementations, each of which is built for a different body type.

The conversion to one of these five JMS message types occurs when the exchange reaches a JMS producer; said another way, it happens when the exchange reaches a route node like this:

```
to("jms:jmsDestinationName")
```

At this point, Camel examines the body type and determines

which JMS message to create. This newly created JMS message is then sent to the JMS destination specified.

Table 6.6 shows what body types are mapped to JMS messages.

Table 6.6 When sending messages to a JMS destination, Camel body types are mapped to specific JMS message types

Camel body type	JMS message type
String, org.w3c.dom.Node	TextMessage
byte[], java.io.File, java.io.Reader, java.io.InputStream, java.nio.ByteBuffer	BytesMessage
java.util.Map	MapMessage
java.io.Serializable	ObjectMessage

Another conversion happens when consuming a message from a JMS destination. **Table 6.7** shows the mappings in this case.

Table 6.7 When receiving messages from a JMS destination, JMS message types are mapped to Camel body types

JMS message type	Camel body type
TextMessage	String
BytesMessage	byte[]
MapMessage	java.util.Map
ObjectMessage	Object
StreamMessage	No mapping occurs

Although this automatic message mapping allows you to fully use Camel's transformation and mediation abilities, you may sometimes need to keep the JMS message intact. An obvious reason is to increase performance—not mapping every message means it takes less time for each message to be processed.

Another reason could be that you're storing an object type that doesn't exist on Camel's classpath. In this case, if Camel tried to

deserialize the object, it would fail when finding the class.

TIP You can also implement your own custom Spring `org.springframework.jms.support.converter.MessageConverter` by using the `messageConverter` option.

To disable message mapping for body types, set the `mapJmsMessage` URI option to `false`.

HEADER MAPPING

Headers in JMS are even more restrictive than body types. In Camel, a header can be named anything that will fit in a Java string, and its value can be any Java object. This presents a few problems when sending to and receiving from JMS destinations.

These are the restrictions in JMS:

- Header names that start with `JMS` are reserved; you can't use these header names.
- Header names must be valid Java identifiers.
- Header values can be any primitive type and their corresponding object types. These include `boolean`, `byte`, `short`, `int`, `long`, `float`, and `double`. Valid object types include `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `String`.

To handle these restrictions, Camel does several things. First, any headers that you set starting with `JMS` are dropped before sending to a JMS destination. Camel also attempts to convert the header names to be JMS-compliant. Any period (.) characters are replaced by `_DOT_`, and any hyphens (-) are replaced with `_HYPHEN_`. For example, a header named `org.apache.camel.Test-Header` would be converted to `org_DOT_apache_DOT_camel_DOT_Test_HYPHEN_Header` before being sent to a JMS destination. If this message is consumed by a Camel route at some point down the line, the header name will be converted back.

To conform to the JMS specification, Camel drops any header that has a value not listed in the list of primitives or their corresponding object types. Camel also allows `CharSequence`, `Date`, `BigDecimal`, and `BigInteger` header values, all of which are converted to their string representations to conform to the JMS specification.

You should now have a good grasp of what Camel can do for your JMS messaging applications. Several types of messaging that we've looked at before, such as JMS and FTP, run on top of other protocols. Let's look at using Camel for these kinds of low-level communications.

6.4 Networking: Netty4 component

So far in this chapter, you've seen a mixture of old integration techniques (such as file-based integration) and newer technologies (such as JMS). All these can be considered essential in any integration framework. Another essential mode of integration is using low-level networking protocols, such as the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). Even if you haven't heard of these protocols before, you've definitely used them—protocols such as email, FTP, and HTTP run on top of TCP.

To communicate over these and other protocols, Camel uses Netty and Apache MINA. Both Netty and MINA are networking frameworks that provide asynchronous event-driven APIs and communicate over various protocols including TCP and UDP. In this section, we use Netty to demonstrate low-level network communication with Camel.

The Netty4 component is located in the `camel-netty4` module of the Camel distribution. You can access this by adding it as a dependency to your Maven POM like this:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty4</artifactId>
```

```

<version>2.20.1</version>
</dependency>

```

The most common configuration options are listed in table 6.8.

Table 6.8 Common URI options used to configure the Netty4 component

Option	Default value	Description
decoder		Specifies the bean used to marshal the incoming message body. It must extend from <code>io.netty.channel.ChannelInboundHandlerAdapter</code> , be loaded into the registry, and referenced using the <code>#beanName</code> style.
decoders		Specifies a list of Netty <code>io.netty.channel.ChannelInboundHandlerAdapter</code> beans to use to marshal the incoming message body. It should be specified as a comma-separated list of bean references (for example, <code>"#decoder1,#decoder2"</code>).
encoder		Specifies the bean used to marshal the outgoing message body. It must extend from <code>io.netty.channel.ChannelOutboundHandlerAdapter</code> , be loaded into the registry, and referenced using the <code>#beanName</code> style.
encoders		Specifies a list of Netty <code>io.netty.channel.ChannelOutboundHandlerAdapter</code> beans to use to marshal the outgoing message body. It should be specified as a comma-separated list of bean references (for example, <code>"#encoder1,#encoder2"</code>).
textline	false	Enables the <code>textline</code> codec when you're using TCP and no other codec is specified. The <code>textline</code> codec understands bodies that have string content and end with a line delimiter.
delimiter	LINE	Sets the delimiter used for the <code>textline</code> codec. Possible values include <code>LINE</code> and <code>NULL</code> .
sync	true	Sets the synchronous mode of communication. Clients will be able to get a response back from the server.

re qu es tT im eo ut	0	Sets the time in milliseconds to wait for a response from a remote server. By default, there's no time-out.
en co di ng	J V M d ef a ul t	Specifies the <code>java.nio.charset.Charset</code> used to encode the data.

In addition to the URI options, you have to specify the transport type and port you want to use. In general, a Netty4 component URI looks like this,

```
netty4:transport://hostname:port[?options]
```

where `transport` is one of `tcp` or `udp`.

Let's now see how to use the Netty4 component to solve a problem at Rider Auto Parts.

6.4.1 USING NETTY FOR NETWORK PROGRAMMING

Back at Rider Auto Parts, the production group has been using automated manufacturing robots for years to assist in producing parts. What they've been lacking, though, is a way of tracking the whole plant's health from a single location. They have floor personnel manually monitoring the machines. What they'd like to have is an operations center with a single-screen view of the entire plant.

To accomplish this, they've purchased sensors that communicate machine status over TCP. The new operations center needs to consume these messages over JMS. [Figure 6.7](#) illustrates this setup.

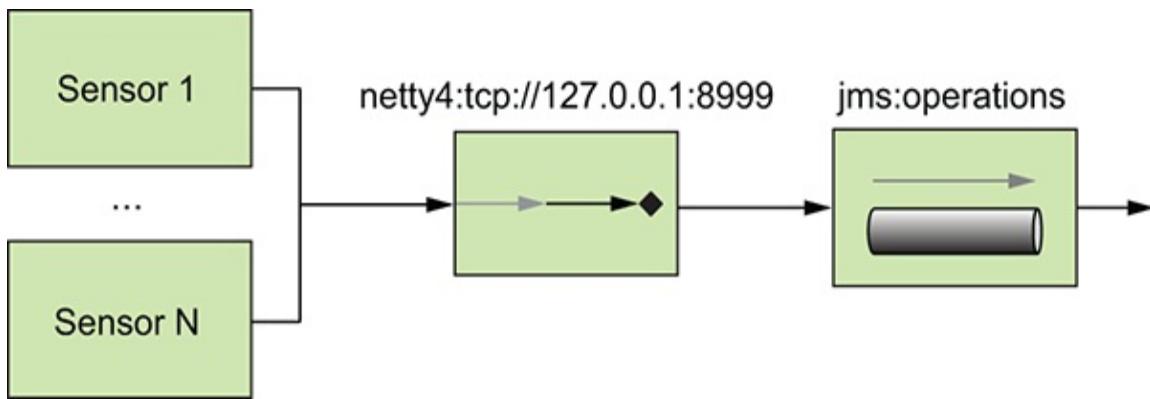


Figure 6.7 Sensors feed status messages over TCP to a server, which then forwards them to a JMS operations queue.

Hand-coding a TCP server such as this wouldn't be a trivial exercise. You'd need to spin up new threads for each incoming socket connection, as well as transform the body to a format suitable for JMS. Not to mention the pain involved in managing the low-level networking protocols.

In Camel, a possible solution is accomplished with a single line:

```
from("netty4:tcp://localhost:8999?
textline=true&sync=false")
    .to("jms:operations");
```

Here you set up a TCP server on port 8999 by using Netty, and it parses messages by using the `textline` codec. The `sync` property is set to `false` to make this route `InOnly`—any clients sending a message won't get a reply back.

NOTE If you set `sync=true`, the route will become `InOut` and return a result to the caller. The `transform` DSL method is handy for setting the return value. See chapter 3 for more details about this.

You may be wondering what a `textline` codec is, and maybe even what a codec is! In TCP communications, a single message payload going out may not reach its destination in one piece. All

will get there, but it may be broken up or fragmented into smaller packets. It's up to the receiver (in this case, the server) to wait for all the pieces and assemble them back into one payload.

A *codec* decodes or encodes the message data into something that the applications on either end of the communications link can understand. As figure 6.8 illustrates, the `textline` codec is responsible for grabbing packets as they come in and trying to piece together a message that's terminated by a specified character.

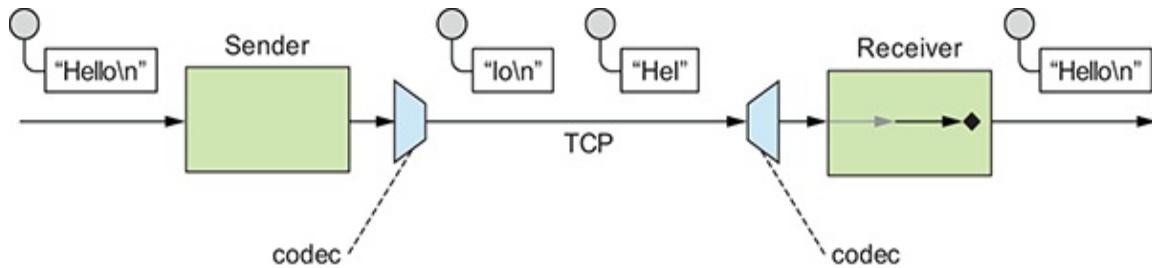


Figure 6.8 During TCP communications, a payload may be broken into multiple packets. A `textline` codec can assemble the TCP packets into a full payload by appending text until it encounters a delimiter character.

This example is provided in the book's source in the `chapter6/netty` directory. Try it by using the following command:

```
mvn test -Dtest=NettyTcpTest
```

OBJECT SERIALIZATION CODEC

If you hadn't specified the `textline` URI option in the previous example, the Netty4 component would've defaulted to using the object serialization codec. This codec takes any `Serializable` Java object and sends its bytes over TCP. This is a handy codec if you aren't sure what payload format to use. If you're using this codec, you also need to ensure that the classes are on the classpath of both the sender and the receiver.

At times your payload will have a custom format that neither `textline` nor object serialization accommodates. In that case, you need to create a custom codec.

6.4.2 USING CUSTOM CODECS

The TCP server you set up for Rider Auto Parts in the previous section has worked out well. Sensors have been sending back status messages in plain text, and you used the Netty `textline` codec to successfully decode them. But one type of sensor has been causing an issue: the sensor connected to the welding machine sends its status back in a custom binary format. You need to interpret this custom format and send a status message formatted like the ones from the other sensors. You can do this with a custom Netty codec.

In Netty, a codec consists of two parts:

- `ChannelOutboundHandler`—This has the job of taking an input payload and putting bytes onto the TCP channel. In this example, the sensor transmits the message over TCP, so you don't have to worry about this too much, except for testing that the server works.
- `ChannelInboundHandler`—This interprets the custom binary message from the sensor and returns a message that your application can understand.

You can specify a custom codec in a Camel URI by using the `encoder/decoder` options (or multiple with the `encoders/decoders` options) and specifying references to instances in the registry.

The custom binary payload that you have to interpret with your codec is 8 bytes in total; the first 7 bytes are the machine ID, and the last byte is a value indicating the status. You need to convert this to the plain-text format used by the other sensors, as illustrated in figure 6.9.

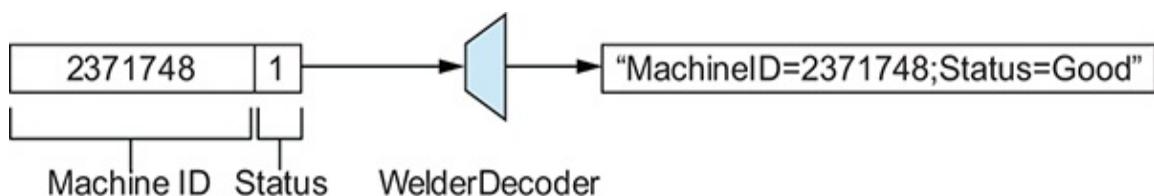


Figure 6.9 The custom welder sensor decoder is used to interpret an 8-byte binary payload and construct a plain-text message body. The first 7 bytes are the machine ID, and the last byte represents a status. In this case, a value of 1 means Good.

Your route looks similar to the previous example:

```
from("netty4://localhost:8998?  
encoder=#welderEncoder&decoder=#welderDecoder&sync=false")  
.to("jms:operations");
```

Note that you need to change the port that it listens on, so as not to conflict with your other TCP server. You also add a reference to the custom codecs loaded into the registry. In this case, the codecs are loaded into a `JndiRegistry` like this:

```
JndiRegistry jndi = ...  
jndi.bind("welderDecoder", new WelderDecoder());  
jndi.bind("welderEncoder", new WelderEncoder());
```

Now that the setup is complete, you can get to the real meat of the custom codec. As you may recall, decoding the custom binary format into a plain-text message was the most important task for this particular application. This decoder is shown in the following listing.

[Listing 6.2](#) The decoder for the welder sensor

```
@ChannelHandler.Sharable  
public class WelderDecoder extends  
MessageToMessageDecoder<ByteBuf> {  
    static final int PAYLOAD_SIZE = 8;  
  
    @Override  
    protected void decode(ChannelHandlerContext ctx,  
ByteBuf msg,  
        List<Object> out) throws Exception {  
        if (msg.isReadable()) {  
            // fill byte array with incoming message  
            byte[] bytes = new byte[msg.readableBytes()];  
            int readerIndex = msg.readerIndex();  
            msg.getBytes(readerIndex, bytes);  
  
            // first 7 bytes are the sensor ID, last is the  
status  
            // and the result message will look something  
like  
            // MachineID=2371748;Status=Good  
            StringBuilder sb = new StringBuilder();
```

```
        sb.append("MachineID=")
            .append(new String(bytes, 0, PAYLOAD_SIZE -
1)).append(";");
    }  
    ①
```

①

Gets first 7 bytes as machine ID

```
        .append("Status=");
        if (bytes[PAYLOAD_SIZE - 1] == '1') {  
    }  
    ②
```

②

Gets last byte as status

```
        sb.append("Good");
    } else {
        sb.append("Failure");
    }
    out.add(sb.toString());
} else {
    out.add(null);
}
}  
}
```

This decoder may look complex, but it's doing only two main things: extracting the first 7 bytes and using that as the machine ID string ①, and checking the last byte for a status of 1, which means Good ②.

To try this example yourself, go to the chapter6/netty directory of the book's source and run the following unit test:

```
mvn test -Dtest=NettyCustomCodecTest
```

Now that you've tried low-level network communications, it's time to interact with one of the most common applications in the enterprise: the database.

6.5 Working with databases: JDBC and JPA components

In pretty much every enterprise-level application, you need to integrate with a database at some point, so it makes sense that Camel has first-class support for accessing databases. Camel has five components that let you access databases in various ways:

- *JDBC component*—Allows you to access JDBC APIs from a Camel route.
- *SQL component*—Allows you to write SQL statements directly into the URI of the component for using simple queries. This component can also be used for calling stored procedures.
- *JPA component*—Persists Java objects to a relational database by using the Java Persistence Architecture.
- *Hibernate component*—Persists Java objects by using the Hibernate framework. This component isn't distributed with Apache Camel because of licensing incompatibilities. You can find it at the camel-extra project (<https://github.com/camel-extra/camel-extra>).
- *MyBatis component*—Allows you to map Java objects to relational databases.

This section covers both the JDBC and JPA components. You can do pretty much any database-related task with them that you can do with the others. For more information on the other components, see the relevant pages on the Camel website's components list (<http://camel.apache.org/components.html>).

Let's look first at the JDBC component.

6.5.1 ACCESSING DATA WITH THE JDBC COMPONENT

The Java Database Connectivity (JDBC) API defines how Java clients can interact with a particular database. It tries to abstract away details about the database being used. To use this component, you need to add the camel-jdbc module to your project:

```
<dependency>
```

```

<groupId>org.apache.camel</groupId>
<artifactId>camel-jdbc</artifactId>
<version>2.20.1</version>
</dependency>

```

The most common URI options are shown in table [6.9](#).

Table 6.9 Common URI options used to configure the JDBC component

Option	Default value	Description
readSize	0	Sets the maximum number of rows that can be returned. The default of 0 causes the readSize to be unbounded.
statement.propertyName		Sets the property with name propertyName on the underlying java.sql.Statement.
useHeadersAsParameters	false	Switches to using a java.sql.PreparedStatement with parameters that are replaced at runtime by message headers. This is a faster and safer alternative to executing a raw java.sql.statement. PreparedStatements are precompiled, and so are faster and aren't susceptible to SQL-injection type attacks.
useJDBC4ColumnNamesAndLabelSemantics	true	Sets the column and label semantics to use. Default is to use the newer JDBC 4 style, but you can set this property to false to enable JDBC 3 style.

The endpoint URI for the JDBC component points Camel to a javax.sql.DataSource loaded into the registry, and, like other components, it allows for configuration options to be set. The URI syntax is as follows:

```
jdbc:dataSourceName[?options]
```

After this is specified, the component is ready for action. But you

may be wondering where the SQL statement is specified.

The JDBC component is a dynamic component in that it doesn't merely deliver a message to a destination but takes the body of the message as a command. In this case, the command is specified using SQL. In EIP terms, this kind of message is called a *command message*. Because a JDBC endpoint accepts a command, it doesn't make sense to use it as a consumer, so you can't use it in a `from DSL` statement. You can still retrieve data by using a `select SQL` statement as the command message. In this case, the query result will be added as the outgoing message on the exchange.

To demonstrate the SQL command-message concept, let's revisit the order router at Rider Auto Parts. In the accounting department, when an order comes in on a JMS queue, the accountant's business applications can't use this data. They can only import data from a database. That means any incoming orders need to be put into the corporate database. Using Camel, a possible solution is illustrated in [figure 6.10](#).

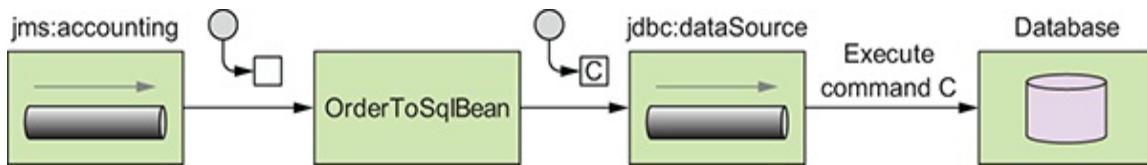


Figure 6.10 A message from the JMS accounting queue is transformed into an SQL command message by the `OrderToSqlBean` bean. The JDBC component then executes this command against its configured data source.

The main takeaway from [figure 6.10](#) is that you're using a bean to create the SQL statement from the incoming message body. This is the most common way to prepare a command message for the JDBC component. You could use the DSL directly to create the SQL statement (by setting the body with an expression), but you have much more control when you use a custom bean.

The route for the implementation of [figure 6.10](#) is simple on the surface:

```
from("jms:accounting")
```

```
.to("bean:orderToSql")
.to("jdbc:dataSource?useHeadersAsParameters=true");
```

Several things require explanation here. First, the JDBC endpoint is configured to load `javax.sql.DataSource` with the name `dataSource` in the registry. The bean endpoint here uses the bean with the name `orderToSql` to convert the incoming message to a SQL statement and to populate headers with information you'll be using in the SQL statement.

The `orderToSql` bean is shown in the following listing.

Listing 6.3 A bean that converts an incoming order to a SQL statement

```
public class OrderToSqlBean {

    public String toSql(@XPath("order/@name") String name,
                        @XPath("order/@amount") int amount,
                        @XPath("order/@customer") String
customer,
                        @Headers Map<String, Object>
outHeaders) {
        outHeaders.put("partName", name);
        outHeaders.put("quantity", amount);
        outHeaders.put("customer", customer);
        return "insert into incoming_orders"
            + "(part_name, quantity, customer) values"
            + " (:?partName, ?:quantity, ?:customer)";
    }
}
```

The `orderToSql` bean uses XPath to parse an incoming order message with a body, something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<order name="motor" amount="1" customer="honda"/>
```

The data in this order is then converted to message headers as follows:

```
partName = 'motor'
quantity = 1
customer = 'honda'
```

You return the same parameterized SQL statement every time:

```
insert into incoming_orders (part_name, quantity, customer)
values (:?partName, ?:quantity, ?:customer)
```

The special `:?` syntax defines the parameter name in the `PreparedStatement` that will be created. The name following `:?` maps to a header name in the incoming message. In our case, the `:?partName` part of the SQL statement will be replaced with the `partName` header value `motor`. The same goes for the other parameters defined. This isn't the default behavior of the JDBC component—you enabled this behavior by setting the `useHeadersAsParameters` URI option to `true`.

This SQL statement becomes the body of a message that will be passed into the JDBC endpoint. In this case, you're updating the database by inserting a new row. You won't be expecting any result back. But Camel will set the `CamelJdbcUpdateCount` header to the number of rows updated. If there were any problems running the SQL command, an `SQLException` would be thrown.

If you were running a query against the database (using a SQL `select` command), Camel would return the rows as a `List<Map<String, Object>>`. Each entry in the `List` is a `LinkedHashMap` that maps the column name to a column value. Camel would also set the `CamelJdbcRowCount` header to the number of rows returned from the query.

To run this example, change to the `chapter6/jdbc` directory of the book's source and run the following command:

```
mvn test -Dtest=JdbcTest
```

Having raw access to the database through JDBC is a must-have ability in any integration framework. At times, though, you need to persist more than raw data; sometimes you need to persist whole Java objects. You can do this with the JPA component, which we'll look at next.

6.5.2 PERSISTING OBJECTS WITH THE JPA COMPONENT

Rider Auto Parts has a new requirement: instead of passing around XML order messages, management would like to adopt a POJO model for orders.

A first step would be to transform the incoming XML message into an equivalent POJO form. In addition, the order persistence route in the accounting department would need to be updated to handle the new POJO body type. You could manually extract the necessary information as you did for the XML message in [listing 6.3](#), but a better solution exists for persisting objects.

The Java Persistence API (JPA) is a wrapper layer on top of object-relational mapping (ORM) products such as Hibernate, OpenJPA, EclipseLink, and the like. These products map Java objects to relational data in a database, which means you can save a Java object in your database of choice, and load it up later when you need it. This is a powerful ability, and it hides many details. Because this adds quite a bit of complexity to your application, plain JDBC should be considered first to see if it meets your requirements.

To use the JPA component, you need to add the camel-jpa module to your project:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jpa</artifactId>
  <version>2.20.1</version>
</dependency>
```

You also need to add JARs for the ORM product and database you're using. The examples in this section use OpenJPA and the Apache Derby database, so you need the following dependencies as well:

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derby</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.openjpa</groupId>
  <artifactId>openjpa-persistence-jdbc</artifactId>
</dependency>
```

The JPA component has URI options, many of which can be applied only to either a consumer or producer endpoint. The URI options are shown in table 6.10.

Table 6.10 Common URI options used to configure the JPA component

Option	Consumer/producer mode	Default value	Description
persistenc eUnit	Both	c a m e l	Specifies the JPA persistence unit name used.
transactio nManager	Both		Sets the transaction manager to be used. If transactions are enabled and this property isn't specified, Camel will use a JpaTransactionManager.
maximumRes ults	Consu mer	-1	Specifies the maximum number of objects to be returned from a query. The default of -1 means an unlimited number of results.
maxMessage sPerPoll	Consu mer	0	Sets the maximum number of objects to be returned during a single poll. The default of 0 means an unlimited number of results.
consumeLoc kEntity	Consu mer	true	Specifies whether the entities in the database will lock while they're being consumed by Camel. By default, they lock.
consumeDelete	Consu mer	true	Specifies whether the entity should be deleted in the database after it's consumed.
consumer.delay	Consu mer	500	Sets the delay in milliseconds between each poll.
consumer.i nitialDelay	Consu mer	1000	Sets the initial delay in milliseconds before the first poll.

y			
consumer.query	Consumer		Sets the custom SQL query to use when consuming objects.
consumer.namedQuery	Consumer		References a named query to consume objects.
consumer.nativeQuery	Consumer		Specifies a query in the native SQL dialect of the database you're using. This isn't very portable, but it allows you to take advantage of features specific to a particular database.
flushOnSend	Producer	true	Causes objects that are sent to a JPA producer to be immediately persisted to the underlying database. Otherwise, they may stay in memory with the ORM tool until it decides to persist.

A requirement in JPA is to annotate any POJOs that need to be persisted with the `javax.persistence.Entity` annotation. The term *entity* is borrowed from relational database terminology and roughly translates to an *object* in object-oriented programming. Your new POJO order class needs to have this annotation if you want to persist it with JPA. The new order POJO is shown in the following listing.

Listing 6.4 An annotated POJO representing an incoming order

`@Entity`

Required annotation for objects to be persisted

```
public class PurchaseOrder implements Serializable {
    private String name;
    private double amount;
    private String customer;

    public PurchaseOrder() {
    }
    public double getAmount() {
```

```

        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setCustomer(String customer) {
        this.customer = customer;
    }
    public String getCustomer() {
        return customer;
    }
}

```

This POJO can be created from the incoming XML order message easily with a message translator, as shown in chapter 3. For testing purposes, you can use a producer template to send a new PurchaseOrder to the accounting JMS queue, like so:

```

PurchaseOrder purchaseOrder = new PurchaseOrder();
purchaseOrder.setName("motor");
purchaseOrder.setAmount(1);
purchaseOrder.setCustomer("honda");

template.sendBody("jms:accounting", purchaseOrder);

```

Your route from section 6.5.1 is now a bit simpler. You send directly to the JPA endpoint after an order is received on the queue:

```

from("jms:accounting").to("jpa:camelaction.PurchaseOrder");

```

Now that your route is in place, you have to configure the ORM tool. This is by far the most configuration you'll have to do when using JPA with Camel. As we've mentioned, ORM tools can be complex.

There are two main bits of configuration: hooking the ORM tool's entity manager up to Camel's JPA component, and

configuring the ORM tool to connect to your database. For demonstration purposes here, we use Apache OpenJPA, but you could use any other JPA-compliant ORM tool.

The beans required to set up the OpenJPA entity manager are shown in the following listing.

Listing 6.5 Hooking up the Camel JPA component to OpenJPA

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-
beans.xsd">
    <bean id="jpa" ①
```

1

Hooks JPA component up to entity manager

```
        class="org.apache.camel.component.jpa.JpaComponent">
            <property name="entityManagerFactory"
ref="entityManagerFactory"/>
        </bean>
    <bean id="entityManagerFactory" ②
```

2

Creates entity manager

```
        class="org.springframework.orm.jpa.LocalEntityManagerFactor
yBean">
            <property name="persistenceUnitName" value="camel"/>
            <property name="jpaVendorAdapter" ref="jpaAdapter"/>
        </bean>
    <bean id="jpaAdapter" ③
```

3

Uses OpenJPA and Apache Derby database

```
class="org.springframework.orm.jpa.vendor.OpenJpaVendorAdapter">
    <property name="databasePlatform"
        value="org.apache.openjpa.jdbc.sql.DerbyDictionary"/>
    <property name="database" value="DERBY"/>
</bean>
<bean id="transactionTemplate" ④
```

④

Allows JPA component to participate in transactions

```
class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager">
        <bean
class="org.springframework.orm.jpa.JpaTransactionManager">
            <property name="entityManagerFactory"
ref="entityManagerFactory"/>
        </bean>
    </property>
</bean>
</beans>
```

This Spring beans file does numerous things to set up JPA. First, it creates a Camel JpaComponent and specifies the entity manager to be used ①. This entity manager ② is then hooked up to OpenJPA and the Derby order database ③. It also sets up the entity manager so it can participate in transactions ④.

There's one more thing left to configure before JPA is up and running. When the entity manager was created in the preceding listing ②, you set the persistenceUnitName to camel. This persistence unit defines what entity classes will be persisted, as well as the connection information for the underlying database. In JPA, this configuration is stored in the persistence.xml file in the META-INF directory on the classpath. The following listing shows the configuration required for your application.

[Listing 6.6 Configuring the ORM tool with the persistence.xml file](#)

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="1.0">
    <persistence-unit name="camel" transaction-
    type="RESOURCE_LOCAL">
        <class>camelaction.PurchaseOrder</class>
```

1

1

Lists entity classes to be persisted

```
<properties>
    <property name="openjpa.ConnectionDriverName"
```

2

2

Provides database connection information

```
        value="org.apache.derby.jdbc.EmbeddedDriver"/>
        <property name="openjpa.ConnectionURL"
        value="jdbc:derby:memory:order;create=true"/>
        <property name="openjpa.ConnectionUserName"
value="sa"/>
        <property name="openjpa.ConnectionPassword"
value=""/>
        <property name="openjpa.jdbc.SynchronizeMappings"
value="buildSchema"/>
    </properties>
</persistence-unit>
</persistence>
```

You need to be aware of two main points in this listing. First, classes that you need persisted need to be defined here 1, and there can be more than one `class` element. Also, if you need to connect to another database or otherwise change the connection information to the database, you need to do so here 2.

Now that all the setup is complete, your JPA route is complete. To try this example, browse to the chapter6/jpa directory and run the `JpaTest` test case with this Maven command:

```
mvn test -Dtest=JpaTest
```

This example sends a `PurchaseOrder` to the accounting queue and then queries the database to make sure the entity class was

persisted.

Manually querying the database via JPA is a useful ability, especially in testing. In `JpaTest`, the query was performed like so:

```
JpaEndpoint endpoint =  
context.getEndpoint("jpa:camelaction.PurchaseOrder",  
JpaEndpoint.class);  
EntityManager em =  
endpoint.getEntityManagerFactory().createEntityManager();  
  
List list = em.createQuery(  
    "select x from camelaction.PurchaseOrder  
x").getResultList();  
  
assertEquals(1, list.size());  
assertInstanceOf(PurchaseOrder.class, list.get );  
  
em.close();
```

First, you create an `EntityManager` instance by using the `EntityManagerFactory` from the `JpaEndpoint`. You then search for instances of your entity class in the database by using JPQL, which is similar to SQL but deals with JPA entity objects instead of tables. A simple check is then performed to make sure the object is the right type and that there's only one result.

Now that we've covered accessing databases, and messaging that can span the entire web, we're going to shift our attention to communication within the JVM.

6.6 In-memory messaging: Direct, Direct-VM, SEDA, and VM components

Having braved so many of Camel's messaging abilities in this chapter, you might think there couldn't be more. Yet there's still another important messaging topic to cover: in-memory messaging.

Camel provides four main components in the core to handle in-memory messaging. For synchronous messaging, there are the Direct and Direct-VM components. For asynchronous

messaging, there are the SEDA and VM components. The only difference between Direct and Direct-VM is that the Direct component can be used for communication within a single CamelContext, whereas the Direct-VM component is a bit broader and can be used for communication within a JVM. If you have two camelContexts loaded into an application server, you can send messages between them by using the Direct-VM component. Similarly, the only difference between SEDA and VM is that the VM component can be used for communication within a JVM.

NOTE For more information on staged event-driven architecture (SEDA) in general, see https://en.wikipedia.org/wiki/Staged_event-driven_architecture.

Let's look first at the Direct components.

6.6.1 SYNCHRONOUS MESSAGING WITH DIRECT AND DIRECT-VM

The Direct component is about as simple as a component can get, but it's extremely useful. It's probably the most common Camel endpoint you'll see in a route.

A direct endpoint URI looks like this:

```
direct:endpointName
```

Table 6.11 lists the only three URI options.

Table 6.11 Common URI options used to configure the Direct and Direct-VM components

Option	Default value	Description
block	false	Causes producers sending to a direct endpoint to block until there are active consumers. Note Camel 2.21 changes this default value to true.

timeout	300 00	The time-out in milliseconds for blocking the producer when there were no active consumers.
failIf NoCons umers	true	Specifies whether to throw an exception when a producer tries to send to an endpoint with no active consumers. Used only when block is set to false.

What does this give you? The Direct component lets you make a synchronous call to a route or, conversely, expose a route as a synchronous service.

To demonstrate, say you have a route that's exposed by a direct endpoint as follows:

```
from("direct:startOrder")
    .to("cxf:bean:orderEndpoint");
```

Sending a message to the `direct:startOrder` endpoint invokes a web service defined by the `orderEndpoint` CXF endpoint bean. Let's also say that you send a message to this endpoint by using `ProducerTemplate`:

```
String reply =
    template.requestBody("direct:startOrder", params,
String.class);
```

`ProducerTemplate` creates a `Producer` under the hood that sends to the `direct:startOrder` endpoint. In most other components, some processing happens between the producer and the consumer. For instance, in a JMS component, the message could be sent to a queue on a JMS broker. With the Direct component, the producer *directly* calls the consumer. And by *directly*, we mean that in the producer there's a method invocation on the consumer. The only overhead of using the Direct component is a method call!

This simplicity and minimal overhead make the Direct component a great way of starting routes and synchronously breaking up routes into multiple pieces. But even though using the Direct component carries little overhead, its synchronous nature doesn't fit well with all applications. If you need to

operate asynchronously, you need the SEDA or VM components, which we'll look at next.

6.6.2 ASYNCHRONOUS MESSAGING WITH SEDA AND VM

As you saw in the discussion of JMS earlier in the chapter (section 6.3), using message queuing as a means of sending messages has many benefits. You also saw that a routing application can be broken into many logical pieces (routes) and connected using JMS queues as bridges. But using JMS for this purpose in an application on a single host adds unnecessary complexity for some use cases.

If you want to reap the benefits of asynchronous messaging, but you aren't concerned with JMS specification conformance or the built-in reliability that JMS provides, you may want to consider an in-memory solution. By ditching the specification conformance and any communications with a message broker (which can be costly), an in-memory solution can be much faster. Note that there's no message persistence to disk, as in JMS, so you run the risk of losing messages in the event of a crash; your application should be tolerant of losing messages.

Camel provides two in-memory queuing components: SEDA and VM. They both share the options listed in table [6.12](#).

Table 6.12 Common URI options used to configure the SEDA and VM components

Option	Default value	Description
size	Integer.MAX_VALUE	Sets the maximum number of messages the queue can hold.
concurrentConsumers	1	Sets the number of threads servicing incoming exchanges. Increase this number to process more exchanges concurrently.

<code>waitForTaskToComplete</code>	<code>IfReplyExpected</code>	Specifies whether the client should wait for an asynchronous task to complete. The default is to wait only if it's an <code>Inout</code> MEP. Other values include <code>Always</code> and <code>Never</code> .
<code>timeout</code>	30000	Sets the time in milliseconds to wait for an asynchronous send to complete. A value less than or equal to 0 disables the timeout.
<code>multipleConsumers</code>	false	Specifies whether to allow the SEDA queue to have behavior like a JMS topic (a publish-subscribe style of messaging).

One of the most common uses for SEDA queues in Camel is to connect routes to form a routing application. For example, recall the example presented in section 6.3.1 in which you used a JMS topic to send copies of an incoming order to the accounting and production departments. In that case, you used JMS queues to connect your routes. Because the only parts that are hosted on separate hosts are the accounting and production queues, you can use SEDA queues for everything else. This new, faster solution is illustrated in [figure 6.11](#).

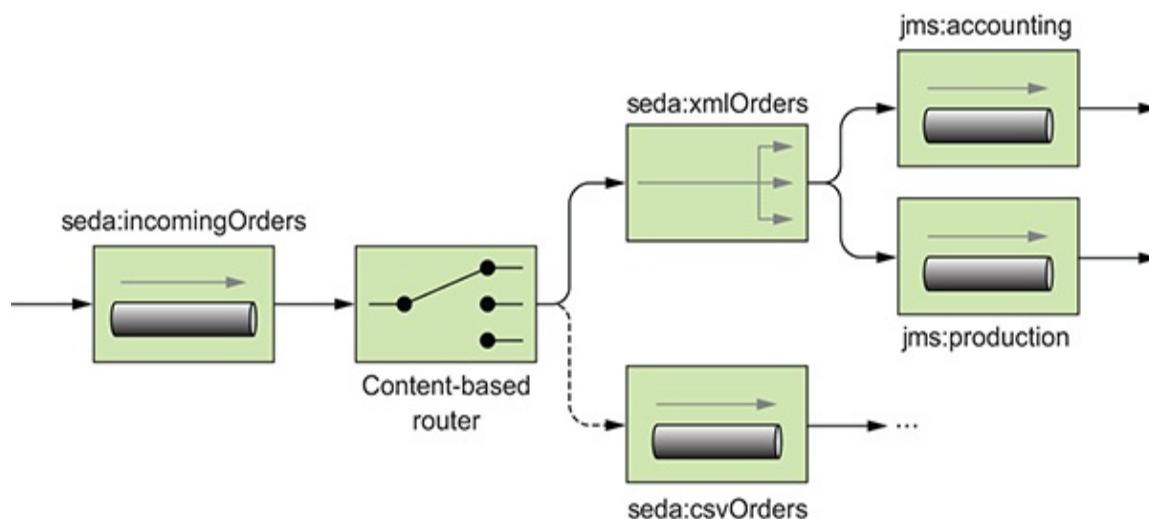


Figure 6.11 SEDA queues can be used as a low-overhead replacement for JMS when messaging is within `CamelContext`. For messages being sent to other hosts, JMS can be used. In this case, all order routing is done via SEDA until the order needs to go to the accounting and production departments.

Any JMS messaging that you were doing within `CamelContext` could be switched over to SEDA. You still need to use JMS for

the accounting and production queues, because they're located in physically separate departments.

You may have noticed that the JMS `xmlOrders` topic has been replaced with a SEDA queue in [figure 6.11](#). In order for this SEDA queue to behave like a JMS topic (using a publish-subscribe messaging model), you need to set the `multipleConsumers` URI option to `true`, as shown in the following listing.

Listing 6.7 A topic allows multiple receivers to get a copy of the message

```
from("file:src/data?noop=true")
    .to("seda:incomingOrders"); ❶
```

Orders enter set of routes

```
from("seda:incomingOrders")
    .choice()
        .when(header("CamelFileName").endsWith(".xml"))
            .to("seda:xmlOrders")
```

XML orders are routed to `xmlOrders` topic

```
        .when(header("CamelFileName").regex("^.*(csv|csl)$"))
            .to("seda:csvOrders");
```

```
from("seda:xmlOrders?multipleConsumers=true")
```

Both listening queues get copies

```
    .to("jms:accounting");
from("seda:xmlOrders?multipleConsumers=true") ❸
    .to("jms:production");
```

This example behaves in the same way as [listing 6.1](#), except that it uses SEDA endpoints instead of JMS. To run this example, go

to the chapter6/seda directory in the book's source and run this command:

```
mvn test -Dtest=OrderRouterWithSedaTest
```

This outputs the following on the command line:

```
Accounting received order: message1.xml  
Production received order: message1.xml
```

Why did you get this output? Well, you have a single order file named message1.xml, and it's published to the xmlOrders topic. Both the accounting and production queues are subscribed to the topic, so each receives a copy. The testing routes consume the messages on those queues and output the messages.

So far, you've been kicking off routes either by hand or by consuming from a filesystem directory. How can you kick off routes automatically? Or better yet, how can you schedule a route's execution to occur?

6.7 Automating tasks: Scheduler and Quartz2 components

Often in enterprise projects you need to schedule tasks to occur either at a specified time or at regular intervals. Camel supports this kind of service with the Timer, Scheduler, and Quartz2 components. The Scheduler component is useful for simple recurring tasks, but when you need more control of when things get started, the Quartz2 component is a must. The Timer component can also be used for simple recurring tasks. The difference is the Scheduler component uses the improved Java scheduler API, as opposed to the Timer component which uses the older java.util.Timer API.

This section first presents the Scheduler component and then moves on to the more advanced Quartz2 component.

6.7.1 USING THE SCHEDULER COMPONENT

The Scheduler component comes with Camel's core library and uses `ScheduledExecutor-Service` from the JRE to generate message exchanges at regular intervals. This component supports only consuming, because sending to a scheduler doesn't make sense.

Some common URI options are listed in table [6.13](#).

Table 6.13 Common URI options used to configure the Scheduler component

Option	Description
delay	Specifies the time in milliseconds between generated events.
initialDelay	Specifies the time in milliseconds before the first event is generated.
useFixedDelay	If true, a delay occurs between the completion of one event and the generation of the next. If false, events are generated at a fixed rate based on the delay period without considering completion of the previous event.

As an example, let's print a message stating the time to the console every 2 seconds. The route looks like this:

```
from("scheduler:myScheduler?delay=2000")
    .setBody().simple("Current time is
${header.CamelTimerFiredTime}")
    .to("stream:out");
```

The scheduler URI configures the underlying `ScheduledExecutorService` to have the execution interval of 2,000 milliseconds.

TIP When the value of milliseconds gets large, you can opt

for a shorter notation using the s, m, and h keywords. For example, 2,000 milliseconds can be written as 2s, meaning 2 seconds. 90,000 milliseconds can be written as 1m30s, and so on.

When this scheduler fires an event, Camel creates an exchange with an empty body and sends it along the route. In this case, you're setting the body of the message by using a Simple language expression. The `camelTimerFiredTime` header was set by the Scheduler component; for a full list of headers set, see the online documentation (<http://camel.apache.org/scheduler.html>).

NOTE The scheduler endpoint and corresponding thread can be shared between routes; you just have to use the same scheduler name.

You can run this simple example by changing to the chapter6/scheduler directory of the book's source and running this command:

```
mvn test -Dtest=SchedulerTest
```

You'll see output similar to the following:

```
Current time is Thu Apr 04 13:43:51 NST 2013
```

As you can see, an event was fired every 2 seconds. But suppose you want to schedule a route to execute on the first day of each month. You can't do that easily with the Scheduler component. You need Quartz.

6.7.2 ENTERPRISE SCHEDULING WITH QUARTZ

Like the Scheduler component, the Quartz2 component allows you to schedule the generation of message exchanges. But the Quartz2 component gives you much more control over how this scheduling happens. You can also take advantage of Quartz's

many other enterprise features.

We don't cover all of Quartz's features here—only ones exposed directly in Camel. For a complete look at using Quartz, see the Quartz website: www.quartz-scheduler.org.

The common URI options for the Quartz2 component are listed in table [6.14](#).

Table 6.14 Common URI options used to configure the Quartz2 component

Option	Default value	Description
cron		Specifies a cron expression used to determine when the timer fires.
trigger.repeatCount	0	Specifies the number of times to repeat the trigger. A value of -1 causes the timer to repeat indefinitely.
trigger.repeatInterval	1000	Specifies the interval in milliseconds at which to generate events.
job.propertyName		Sets the property with name <code>propertyName</code> on the underlying Quartz <code>JobDetail</code> .
trigger.propertyName		Sets the property with name <code>propertyName</code> on the underlying Quartz <code>Trigger</code> .

Before you can use the Quartz2 component, you need to add the following dependency to your Maven POM file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-quartz2</artifactId>
  <version>2.20.1</version>
</dependency>
```

Let's now reproduce the scheduler example from the previous section with Quartz. To do this, you can use the `trigger.repeatInterval` option, which is similar to the `delay` option for the Scheduler component. The route looks like this:

```
from("quartz2:myTimer?
trigger.repeatInterval=2000&trigger.repeatCount=-1")
```

```
.setBody().simple("Current time is  
${header.firedTime}")  
.to("stream:out");
```

Although this behaves in the same way as the scheduler example, a few things are going on under the covers.

First, `myTimer` sets the underlying `Trigger` object's name. Timers in Quartz are made up of a `Trigger` and a `JobDetail`. Triggers can also have a group name associated with them, which you can specify by adding the group name to your URI, as follows:

```
quartz2:myGroupName/myTimer?options...
```

If the group name is omitted, as in the previous route, *Camel* is used as the group name. By default, `SimpleTrigger` is created to schedule events.

The `trigger.repeatInterval` and `trigger.repeatCount` properties configured this trigger to fire every 2,000 milliseconds for as long as the application is running (a repeat count of `-1` causes the trigger to repeat indefinitely). You may be thinking that the option names are a bit long, but there's a reason for this. As stated in table 6.14, options starting with `trigger` allow you to set properties on the `Trigger` object. In the case of the `trigger.repeatInterval` URI option, this will call the `setRepeatInterval` method on the `SimpleTrigger` object.

You can similarly set options on `JobDetail` by using properties that start with `job`, followed by a valid property name. For instance, you can set the job name by using the `job.name` URI option.

You can run this simple example by changing to the `chapter6/quartz` directory of the book's source and running this command:

```
mvn test -Dtest=QuartzTest
```

USING CRON TRIGGERS

So far, you've replaced the Scheduler component example with a functionally equivalent Quartz-based example. How would you schedule something more complex, such as kicking off a route on the first day of each month? The answer is by using cron expressions. Readers familiar with Linux or UNIX probably have heard of the cron scheduling application. Quartz allows you to use scheduling syntax similar to the venerable cron application.

A *cron* expression is a string consisting of six or seven fields separated by whitespace. Each field denotes a date or range of dates. The structure of a cron expression is as follows:

```
<Seconds> <Minutes> <Hours> <Day of Month> <Month> <Day of week> <Year>
```

These accept numeric values (and optional textual ones) for the times you want a trigger to fire. More information on cron expressions can be found on the Quartz website (www.quartz-scheduler.org/documentation/quartz-2.x/tutorials/crontrigger).

The cron expression for occurring on the first day of each month at 6:00 a.m. is the following:

```
0 0 6 1 * ?
```

In this expression, the third digit denotes the hour at which to execute, and the fourth digit is the day of the month. You can also see that a star is placed in the month column so that every month will be triggered.

Setting up Quartz to use cron triggers in Camel is easy. You just use the `cron` option and make sure to replace all whitespace with plus characters (+). Your URI becomes the following:

```
quartz2:firstDayOfTheMonth?cron=0+0+6+1+*+?
```

Using this URI in a route causes a message exchange to be generated (running the route) on the first day of each month.

To try an example using a cron trigger, browse to the `chapter6/quartz` directory and run the `QuartzCronTest` test case with this Maven command:

```
mvn test -Dtest=QuartzCronTest
```

You should be able to see now how the scheduling components in Camel can allow you to execute routes at specified times. This is an important ability in time-sensitive enterprise applications.

6.8 Working with email

It's hard to think of a more pervasive technology than email in the enterprise. Modern businesses, for better or worse, run on email. Compared to the other communication mechanisms mentioned thus far, email is certainly different. Whereas other components are primarily used to communicate with other automated services, email is most often used to communicate with people. A retail application may need to notify a customer that an order has been shipped, for instance. Or maybe a back-end system needs to send an alert to a system administrator about a failure. These are all ideal for email messaging. Sure, technically you can implement inter-application messaging with email if you want to, but that isn't the most efficient way to do things.

Camel provides several components to work with email:

- *Mail component*—This is the primary component for sending and receiving email in Camel.
- *AWS-SES component*—Allows you to send email by using the Amazon Simple Email Service (SES).
- *GoogleMail component*—Gives you access to Gmail via the Google Mail Web API.

This section covers the mail component. For more information on the other components, see the relevant pages on the Camel website's components list (<http://camel.apache.org/components.html>).

Let's first take a look at sending email.

6.8.1 SENDING MAIL WITH SMTP

Whenever you send an email, you're using the Simple Mail Transfer Protocol (SMTP) under the hood. In Camel, an SMTP URI looks like this:

```
[smtp|stmps]://[username@]host[:port][?options]
```

You'll first notice that you can select to secure your mail transfer with SSL by specifying a scheme of `stmps` instead of `smtp`. The host is the name of the mail server that will be sending the message, and `username` is an account on that server. The value of `port` defaults to 25 for SMTP, and 465 for SMTPS, and can be overridden if needed. The most common URI options are shown in table 6.15.

Table 6.15 Common URI options used to configure the Mail component

Option	Default value	Description
<code>password</code>		The password of the user account corresponding to <code>username</code> in the URI.
<code>subject</code>		Sets the subject of the email being sent. You can override this value by setting a <code>Subject</code> message header.
<code>from</code>	<code>camel@localhost</code>	Sets what email address will be used for the <code>From</code> field of the email being sent.
<code>to</code>	<code>username@host</code>	The email address you're sending to. Multiple addresses must be separated by commas.
<code>cc</code>		The carbon copy (CC) email address you're sending to. Multiple addresses must be separated by commas.

To use this component, you need to add the `camel-mail` module to your project:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mail</artifactId>
  <version>2.20.1</version>
</dependency>
```

Sending an email is then simple. For example, let's say Jon wants to send an email to Claus about Camel. Using ProducerTemplate, you could do something like this:

```
template.sendBody(  
    "smtp://jon@localhost?  
password=secret&to=claus@localhost",  
    "Yes, Camel rocks!");
```

Here, you're logging into an email server on localhost port 25 with username `jon` and password `secret`. You're sending to the email address `claus@localhost`. Because you didn't specify a `From` option, the `From` email field is defaulted to `camel@localhost`. Now the string you passed as a message body will become the body of the email. Typically, though, you'd like to have a subject for the email. You could add a `subject` option to the URI or add a `subject` header to the message. Let's try to add a `subject` header:

```
template.sendBodyAndHeader(  
    "smtp://jon@localhost?  
password=secret&to=claus@localhost",  
    "Yes, Camel rocks!",  
    "subject", "Does Camel rock?");
```

Again, you use ProducerTemplate to send, but this time you pass along a message header to set the subject as well.

The `claus@localhost` email address could be checked in a GUI mail client by a person, or you could get Camel to consume this for you. Let's take a look at how to do this next.

6.8.2 RECEIVING MAIL WITH IMAP

In Camel, you can retrieve email by using either the Internet Message Access Protocol (IMAP) or POP3. IMAP is the preferred protocol to access email. It was developed as an alternative to POP3 and is more feature rich. To consume email messages by using IMAP, you need to use a URI like this:

```
[imap|imaps]://[username@]host[:port][?options]
```

As with the SMTP component, you can select to secure your mail transfer with SSL by specifying a scheme of `imaps` instead of `imap`. The `host` is the name of the mail server that has the mail you want to consume, and `username` is an account on that server. The value of `port` defaults to 143 for IMAP, and 993 for IMAPS, and can be overridden if needed. The most common URI options are shown in table 6.16.

Table 6.16 Common URI options used to configure the Mail component

Option	Default value	Description
<code>password</code>		The password of the user account corresponding to <code>username</code> in the URI.
<code>delay</code>	60000	Delay between polling the mail server for more emails.
<code>delete</code>	false	If true, the email will be deleted after processing.
<code>folderName</code>	INBOX	The mail folder to poll for messages.
<code>unseen</code>	true	If true, consumes only new messages.

As with the SMTP component, you first need to add the camel-mail module to your project.

Let's say you want to use IMAP to receive the email Jon sent via SMTP in the preceding section. The route looks like this:

```
from("imap://claus@localhost?
password=secret").to("mock:result");
```

Here, you're logging into an email server on localhost port 143 with username `claus` and password `secret`. You're polling the INBOX folder for new messages by using the default polling interval of 60 seconds. To try this example, browse to the `chapter6/mail` directory and run the `MailTest` test case with this Maven command:

```
mvn test -Dtest=MailTest
```

That covers the basics of sending and receiving email using Camel. Many more options are available. You can find them all in the online documentation (<http://camel.apache.org/mail.html>).

6.9 Summary and best practices

Congratulations on making it through the barrage of components covered in this chapter. By now, you should have a good understanding of how to use them in your own applications.

Here are some of the key ideas you should take away from this chapter:

- *There are tons of Camel components*—One of the great things about Camel is its extensive component library. You can rest easy knowing that most things you'll ever need to connect to are covered by Camel.
- *The Camel website has documentation on all components available*—We could cover only the most widely used and important components in this book, so if you need to use one of the many other components, documentation is available at <http://camel.apache.org/components.html>.
- *Camel's component model allows for your own extensions*—We briefly touched on how components are resolved at runtime in Camel. Camel imposes no restrictions on where your components come from, so you can easily write your own (as described in chapter 8) and include it in your Camel application.
- *Don't write to files manually; use the File and FTP components*—Camel's File and FTP components have many options that suit most file-processing scenarios. Don't reinvent the wheel—use what Camel has to offer.
- *Use the JMS component for asynchronous messaging with JMS*—Camel makes it easy to send messages to and receive them from JMS providers. You no longer have to write dozens of lines

of JMS API calls to send or receive a simple message.

- *Use the Netty4 component for network communications*—Network programming can be difficult, given the low-level concepts you need to deal with. The Netty4 component handles these details for you, making it easy to communicate over network protocols such as TCP and UDP.
- *Hook your routes into databases by using the JDBC and JPA components*—The JDBC component allows you to access databases by using tried-and-true SQL, whereas the JPA component is all about persisting Java objects into databases.
- *Use in-memory messaging when reliability isn't a concern but speed is*—Camel provides four choices for in-memory messaging: the Direct, Direct-VM, SEDA, and VM components.
- *Kick off routes at specified intervals by using the Quartz2 or Scheduler components*—Camel routes can do useful things. Some tasks need to be executed at specified intervals, and the Quartz2 and Scheduler components come into play here.

Components in Camel fill the role of bridging out to other transports, APIs, and data formats. They're also the on- and off-ramps to Camel's routing abilities.

Next up in chapter 7, is a topic that has been a buzzword for a while now: *microservices*. Camel wasn't designed from the ground up with this in mind, but it has certainly become a useful way of building these small single-purpose services.

Table 6.2 Components in the camel-core module (continued)

Part 3

Developing and testing

Camel is ideal for building microservice applications, which is the topic of chapter 7. The chapter has many examples that demonstrate how to use Camel with popular microservice runtimes such as Spring Boot and WildFly Swarm. You'll see how to build small discrete Camel microservices that, when combined, solve a real-life business case. The Circuit Breaker EIP pattern is an important pattern typically used with microservices in distributed systems, and you'll find plenty of deep coverage of this pattern in chapter 7.

In chapter 8, we'll discuss a topic you could really read right after part 1: how to develop new Camel projects. In this chapter, we'll show you how to create new Camel projects, which could be Camel applications, custom components, or custom data formats. You'll also learn techniques of debugging your Camel routes.

In chapter 9, we'll take a look at another important topic in application development: testing. We'll look at the testing facilities shipped with Camel. You can use these features for testing your own Camel applications or applications based on other stacks.

RESTful web services have become a ubiquitous protocol in recent years and are the topic of chapter 10. You'll hear about the many Camel components that support RESTful services, and we'll walk you through Camel's Rest DSL, which allows you to design APIs in an easy and intuitive manner—*the Camel way*. Speaking of APIs, you'll also see how you can document your

APIs using Swagger with Camel.

7

Microservices

This chapters covers

- Microservices overview and characteristics
- Developing microservices with Camel
- WildFly Swarm and Camel
- Spring Boot and Camel
- Designing for failures
- Netflix Hystrix circuit breakers

We wanted to bring you the most up-to-date material, so, because the IT industry is evolving constantly, we decided to wait and write chapters 7 and 18 last. This chapter covers using microservices with Camel, and chapter 18 is a natural continuation that goes even further and takes the microservices world into the cloud by using container technology such as Docker and Kubernetes. This chapter stays down-to-earth and is mainly focused on running locally on a single computer or laptop.

We initially wrote a lengthy chapter introduction to talk about how the software is eating the world, how businesses are being disrupted, and how new emerging businesses are doing business in a faster and more agile way, where microservices are instrumental. But we felt there are much better authors who can

explain all that, and we mention two of them at the start of section 7.1.

The first section covers six microservices characteristics that you should have in mind when reading the remainder of this chapter.

Section 7.2 jumps into action by showing you how to build microservices with Camel by using various JVM frameworks, starting with plain, standalone Camel and moving all the way to using popular Java microservice runtimes such as Spring Boot and WildFly Swarm.

Section 7.3 covers how microservices can call each other. The section examines a use case at Rider Auto Parts as you build a prototype consisting of six microservices.

The first part of section 7.4 covers strategies you can use for building fault-tolerant microservices. The second part continues the Rider Auto Parts use case and walks you through improving those six microservices to become a fully functional fault-tolerant system. As part of the exercise, you'll learn how to use the popular Hystrix circuit breaker from the Netflix OSS stack and see firsthand how elegantly Camel integrates with it.

The chapter ends on a high note with some eye candy of a live graph that charts the state of the circuit breakers.

Let's start from the beginning and set the stage so you can understand the forces at play.

7.1 Microservices overview

The more abstract and fluffy a concept seems, the harder it is to explain and get everybody on board so they have a similar view of the landscape. Microservices is no exception. Don't mistake microservices as only a technology discussion. It's equal parts organizational structure, culture, and human forces.

Microservices aren't a new technical invention, like an operating system or programming language. Microservices are a

style of software system that emerges when you stick to a set of design principles and methodologies. This section focuses on the technical characteristics of microservices. If you’re interested in the nontechnical side, we recommend (among others) the following:

- *Building Microservices* by Sam Newman (O’Reilly, 2015)
- *Microservices for Java Developers* by Christian Posta, a free ebook published by O’Reilly (<https://developers.redhat.com/promotions/microservices-for-java-developers/>)

In this section, we cover six microservices characteristics that we consider most relevant to this book at the time of writing. In addition to learning about these characteristics and the common practices used for implementing microservices, you’ll see the role Camel plays.

7.1.1 SMALL IN SIZE

A fundamental principle of a microservice is hinted at by its name: a *microservice* is small (micro) in size. The mantra of a microservice is said *to be small, nimble, do one thing only, and do that well*. This is reinforced by Robert C. Martin’s definition of the *single responsibility principle*, which states: “Gather together those things that change for the same reason, and separate those things that change for different reasons.”

Microservices takes the same approach by focusing on service and business boundaries, making it obvious where the code lives for a given functionality. And by keeping the service focused on an explicit boundary, we avoid any temptation for it to grow too large, with all the complexities that can introduce. How to measure the size of a microservice is debatable—do you use the number of code lines, the number of classes, or some other measurement? A good measure of the size of a microservice is that a single person should be able to manage all of it in their head.

Camel shines in this area. Camel applications are inherently

small. For example, a Camel application that receives events over HTTP or messaging, transforms and operates on the data, and sends a response back can be about 50 to 100 lines of code. That's small enough to fit into the head of one person who can build end-to-end testing and do code refactoring. And if the microservice is no longer needed, the code can be thrown away without losing or wasting hundreds of hours of implementing and coding.

7.1.2 OBSERVABLE

Services should expose information that describes the state of the service and the way the service performs. Running microservices in production should allow easy management and monitoring. A good practice is to provide health checks in your microservices, accessible from known endpoints that the runtime platform uses for constant monitoring. This kind of information is provided out of the box by Camel, and you can make it available over JMX, HTTP, and Java microservice runtimes such as Spring Boot and WildFly Swarm.

TIP Chapter 16 covers details of management and monitoring with Camel.

7.1.3 DESIGNED FOR FAILURE

A consequence of building distributed systems by using microservices is that applications need to be designed so they can tolerate failures of services. Any upstream service can fail for any number of reasons, which imposes upon the client to respond to the failure as gracefully as possible. Because a service can come and go at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore the service on the runtime platform—hence be self-monitoring.

Camel has many EIP patterns to help design with failures. The Dead Letter Channel EIP can ensure that messages aren't lost in

the event of service failures. The error handler can perform redelivery attempts to attempt to mitigate a temporary service failure. The load balancer can balance the service call between multiple machines hosting the same service. The Service Call EIP (covered in chapter 17) can, based on service availability, call a healthy service on a set of machines hosting the same service. The Circuit Breaker EIP can be used to protect unresponsive upstream services from receiving further calls during a period of time while the circuit breaker is open. We cover designing for failures and using the Circuit Breaker EIP in section 7.4.

7.1.4 HIGHLY CONFIGURABLE

A sound practice is to never hardcode or commit your environment-specific configuration, such as usernames and passwords, in your source code. Instead, keep the configuration outside your application in external configuration files or environment variables. A good mantra to help you remember this practice goes like this: *Separate config from code*.

By keeping configuration outside your application, you can deploy the same application to different environments by changing only the configuration and not the source code. With the rise of container platforms such as Docker Swarm, Kubernetes, Apache Mesos, and Cloud Foundry, this has become common practice: you can safely move a Docker image (with your application) between environments (test, staging, pre-production, and production) by applying environment-specific configuration to the Docker image.

Camel applications are highly configurable; you can easily externalize configuration of all your Camel routes, endpoints, and so on. This configuration can then easily be configured per environment without having to recompile your source code or set up credentials, encryption, threading models, and whatnot.

7.1.5 SMART ENDPOINTS AND DUMB PIPES

Over the last couple of decades, we've seen different integration products based on the principles of Enterprise Application

Integration (EAI), enterprise service bus (ESB), and service-oriented architecture (SOA). A common denominator for these products is that they include sophisticated facilities for message routing, mediation, transformation, business rules, and much more. The most complex of these products rely on overly complex protocols such as Web Service Choreography (WS-Choreography), Business Process Execution Language (BPEL), or Business Process Model and Notation (BPMN).

Microservices favor RESTful and lightweight protocols rather than complex protocols such as SOAP. The style used by microservices is *smart endpoints and dumb pipes*. Applications built from microservices aim to be as decoupled and as cohesive as possible. They own their own domain logic and act more like the UNIX shell commands (pipes and filters): receive an event, apply some logic, and return a response.

Camel supports both worlds. You can find components supporting SOAP Web Services, BPMN, and JMS. For microservices, you can find a lot of components supporting REST, as well as a Rest DSL, which makes defining RESTful services much easier (we cover this in chapter 10). There are plenty of components for lightweight messaging, such as Advanced Message Queuing Protocol (AMQP), Amazon Simple Queue Service (SQS), Amazon Simple Notification Service (SNS), Kafka, Message Queue Telemetry Transport (MQTT), gRPC, and Protocol Buffers. And each new Camel release brings more components.

7.1.6 TESTABLE

Let's step back and ask ourselves, why are we using microservices? There may be many reasons for doing so, but a key goal should be delivering services faster into production. By having many more microservices deployed into production, developers and operations teams can have confidence that what's being deployed is working as intended, and can help ensure that testing microservices is faster and easier—and as automated as possible.

Camel has extensive support for testing at different levels, including unit and integration tests. With Camel, you can test your application in isolation by mocking external endpoints, simulating events, and verifying that all is as expected. We devote all of chapter 9 to this.

We'll touch on these microservices characteristics throughout the rest of the chapter.

It's time to leave the background theory, get down to action, and show some code. The next section teaches you how to get Camel riding on the hype of microservice waves.

7.2 *Running Camel microservices*

Which microservice runtimes does Camel support? The answer is *all of them*. Camel is just a library you include in the JVM runtime, and it runs anywhere. This section walks you through running Camel in some of the most popular microservice runtimes:

- *Standalone*—Running just Camel
- *CDI*—Running Camel with CDI
- *WildFly Swarm*—We'll see how Camel runs with the lightweight Java EE server
- *Spring Boot*—Running Camel with Spring Boot

As you can see, there are four runtimes in our bulleted list, so there's a lot to cover. We start with just Camel and then work our way down the list, ending with some of the newer and more eccentric microservice runtimes.

Before continuing, we want to mention that chapter 20 (available online only) covers another microservice framework called Vert.X, which you can use to build reactive applications. Vert.X isn't your typical framework for running Camel applications, but it has potential, so we want to make sure you hear about it.

7.2.1 STANDALONE CAMEL AS MICROSERVICE

Throughout this section, we use a basic Camel example as the basis of a microservice. The service is an HTTP hello service that returns a simple response, as shown in the following Camel route:

```
public class HelloRoute extends RouteBuilder {  
  
    public void configure() throws Exception {  
        from("jetty:http://localhost:8080/hello")  
            .transform().simple("Hello from Camel");  
    }  
}
```

To run this service by using standalone Camel, you follow these three steps:

1. Create a CamelContext.
2. Add the route with the service to the CamelContext.
3. Start the CamelContext.

```
public class HelloCamel {  
  
    public static void main(String[] args) throws Exception {  
        CamelContext context = new DefaultCamelContext();
```

Creates the CamelContext

```
        context.addRoutes(new HelloRoute());
```

Adds the route with the service

```
        context.start();
```

Starts Camel

```
        Thread.sleep(Integer.MAX_VALUE); ①
```

①

Keeps Camel (the JVM) running

```
}
```

You can try this example from the source code in the chapter7/standalone directory by running the following Maven goal:

```
mvn compile exec:java -Pmanual
```

When the example is running, you can access the service from a web browser on <http://localhost:8080/hello>.

Have you noticed a code smell in the `HelloCamel` class? Yeah, at ① we had to use `Thread.sleep` to keep Camel running. Whenever there's a code smell, there's usually a better way with Camel.

RUNNING CAMEL STANDALONE BY USING THE CAMEL MAIN CLASS

Camel provides an `org.apache.camel.main.Main` class, which makes it easier to run Camel standalone:

```
public class HelloMain {  
  
    public static void main(String[] args) throws Exception  
{  
    Main main = new Main();
```

Creates the Main class

```
    main.addRouteBuilder(new HelloRoute());
```

Adds the route with the service

```
    main.run(); ①
```

①

Keeps Camel (the JVM) running

```
}
```

As you can see, running Camel by using the `Main` class is easier. The `run` method ❶ is a blocking method that keeps the JVM running. The `Main` class also ensures that you trigger on JVM shutdown signals and perform a graceful shutdown of Camel so your Camel applications terminate gracefully.

You can try this example from the source code in the `chapter7/standalone` directory by running the following Maven goal:

```
mvn compile exec:java -Pmain
```

To terminate the JVM, press Ctrl-C, and notice from the console log that Camel is stopping.

The `Main` class has additional methods for configuration, such as property placeholders, and for registering beans in the registry. You can find details by exploring the methods available on the main instance.

SUMMARY OF USING STANDALONE CAMEL FOR MICROSERVICES

Running Camel standalone is the simplest and smallest runtime for running Camel—it's just Camel. You can quickly get started, but it has its limits. For example, you need to figure out how to package your application code with the needed JARs from Camel and third-party dependencies, and how to run that as a Java application. You could try to build a fat JAR, but that creates problems if duplicate files need to be merged together. But if you're using Docker containers, you can package your application together in a Docker image, with all the JARs and your application code separated, and still make it easy to run your application. We cover Camel and Docker in chapter 18.

The `Main` class has its limitations, and you may want to go for some of the more powerful runtimes, such as WildFly Swarm or

Spring Boot, which we cover in sections to follow. But first, let's talk about Camel and CDI.

7.2.2 CDI CAMEL AS MICROSERVICE

What is CDI? Contexts and Dependency Injection (CDI) is a Java specification for dependency injection that includes a set of key features:

- *Dependency injection*—Dependency injection of beans.
- *POJOs*—Any kind of Java bean can be used with CDI.
- *Lifecycle management*—Perform custom action on bean creation and destruction.
- *Events*—Send and receive events in a loosely coupled fashion.
- *Extensibility*—Pluggable extensions can be installed in any CDI runtime to customize behavior.

Camel provides support for CDI by the camel-cdi component, which contains a set of CDI extensions that make it easy to set up and bootstrap Camel with CDI.

TIP You can find great CDI documentation at the JBoss Weld project: <http://docs.jboss.org/weld/reference/latest/en-US/html/>.

HELLO SERVICE WITH CDI

The Camel route for the hello microservice can be written using CDI, as shown in the following listing.

Listing 7.1 Camel route using CDI

@Singleton

1

1

Defines scope of bean as Singleton

```
public class HelloRoute extends RouteBuilder {  
  
    public void configure() throws Exception {  
        from("jetty:http://localhost:8080/hello")  
            .transform().simple("Hello from Camel");  
    }  
}
```

As you can see, the Camel route is just regular Java DSL code without any special CDI code. The only code from CDI is the `@Singleton (javax.inject.Singleton)` annotation **1**.

To make running Camel in CDI easier, you can use an `org.apache.camel.cdi.Main` class:

```
public class HelloApplication {  
  
    public static void main(String[] args) throws Exception  
{  
        Main main = new Main();  
        main.run();  
    }  
}
```

You can try this example from the `chapter7/cdi-hello` directory by running the following Maven goal:

```
mvn compile exec:java
```

The hello service can be accessed from a web browser at `http://localhost:8080/hello`. To terminate the JVM, you can press Ctrl-C.

This example is simple, so let's push the bar a little and configure Camel using CDI.

CONFIGURING CDI APPLICATIONS

The code in [listing 7.1](#) returns a reply message that's hardcoded. Let's improve this and externalize the configuration to a properties file. Camel's property placeholder mechanism is a component (`PropertiesComponent`) with the special name `properties`. In CDI, you use `@Produces` and `@Named` annotations to

create and configure beans. You use this to create and configure an instance of `PropertiesComponent`, as shown in the following listing.

Listing 7.2 Configuring Camel property placeholder by using CDI `@Produces`

```
@Singleton  
public class HelloConfiguration {
```

```
    @Produces      ①
```

①

Declares that this method is producing (creating) a bean

```
        @Named("properties")      ②
```

②

... with the name `properties`

```
    PropertiesComponent propertiesComponent() {      ③
```

③

... and of type `PropertiesComponent`

```
        PropertiesComponent component = new  
        PropertiesComponent();  
  
        component.setLocation("classpath:hello.properties");  
        return component;  
    }  
}
```

Here you've created a class named `HelloConfiguration` that you use to configure your application with CDI. It's good practice to have one or more configuration classes separated from your business and Camel routing logic.

To set up Camel's property placeholder, you use a Java method that creates, configures, and returns an instance of

`PropertiesComponent` ③. Notice that the code in this method is plain Java code without using CDI. Only on the method signature do you use CDI annotations, to instruct CDI that this method is able to produce ① a bean of type `PropertiesComponent` with the ID `properties` ②.

This example is shipped with the book's source code in the `chapter7/cdi` directory, which you can try by using the following Maven goal:

```
mvn compile exec:java
```

Let's raise the bar one more time and show you the most-used feature of CDI: using CDI dependency injection in your POJO beans with `@Inject`.

DEPENDENCY INJECTION USING `@Inject`

We've refactored the example to use a POJO bean to construct the reply message of the hello service:

```
@Singleton  
public class HelloBean {  
  
    public String sayHello(@PropertyInject("reply") String  
msg) ①
```

① Injects Camel property placeholder with key `reply` as parameter

```
        throws Exception {  
        return msg + " from " +  
InetAddressUtil.getLocalHostName();  
    }  
}
```

The `HelloBean` class has a single method named `sayHello` that creates the reply message. The method takes one argument as input that has been annotated with Camel's `@PropertyInject` annotation. `@PropertyInject` is configured with the value `reply` ①, which corresponds to the property key. The property placeholder

file should contain this key:

```
reply=Hello from Camel CDI with properties
```

The Camel route for the hello service needs to be changed to use the `HelloBean`; this can be done by dependency-injecting the bean into the `RouteBuilder` class, as shown in the following listing.

Listing 7.3 Injecting bean using CDI `@Inject`

```
@Singleton  
public class HelloRoute extends RouteBuilder {
```

```
    @Inject ①
```

①

Dependency injects the `HelloBean` using `@Inject`

```
        private HelloBean hello;  
  
        public void configure() throws Exception {  
            from("jetty:http://localhost:8080/hello")  
                .bean(hello, "sayHello"); ②
```

②

Calls the method `sayHello` on the injected `HelloBean` instance

```
}
```

Because you want to use the `HelloBean` in the Camel route, you can instruct CDI to dependency-inject an instance of the bean by declaring a field and annotate the field with `@Inject` ①. The bean can then be used in the Camel route as a bean method call ②.

This example is provided with the book's source code in the `chapter7/cdi` directory. With the source code, you see how to use `@Inject` in unit tests with CDI. We've devoted all of chapter 9 to the topic of testing and cover testing CDI in much more detail.

The camel-cdi component provides a specialized dependency injection to inject Camel endpoints in POJOs.

USING @URI TO INJECT CAMEL ENDPOINT

In your POJOs or Camel routes, you may want to inject a Camel endpoint. For example, instead of using string values for Camel endpoints in Camel routes, you can use endpoint instances instead:

```
@Singleton  
public class HelloRoute extends RouteBuilder {  
  
    @Inject  
    private HelloBean hello;  
  
    @Inject @Uri("jetty:http://localhost:8080/hello") ❶
```

❶

Dependency injects Camel endpoint with the given URI

```
private Endpoint jetty;  
  
public void configure() throws Exception {  
    from(jetty) ❷
```

❷

Uses the injected Camel endpoint in the Camel route

```
.bean(hello, "sayHello");  
}  
}
```

In the `HelloRoute` class, we use a field to define the Camel endpoint for the incoming endpoint ❶. Notice that the field is annotated with both `@Inject` (CDI) and `@Uri` (camel-cdi). In the Camel route, the endpoint is used in the form that accepts `org.apache.camel.Endpoint` as a parameter type ❷.

You can also use `@Inject @Uri` to inject `FluentProducerTemplate`, which makes it easy to send a message

to the given endpoint. The following listing uses this in unit testing the example.

Listing 7.4 Using `@Inject @Uri` to inject `FluentProducerTemplate`

```
@RunWith(CamelCdiRunner.class) 1
```

1

Unit testing with camel-cdi

```
public class HelloRouteTest {  
    2  
    @Inject @Uri("jetty:http://localhost:8080/hello")  
  
    private FluentProducerTemplate producer;  
  
    @Test  
    public void testHello() throws Exception {  
        String out = producer.request(String.class); 3  
  
        assertTrue(out.startsWith("Hello from Camel CDI"));  
    }  
}
```

3

Sends `(InOut)` empty message to the endpoint and receives response as `String` type

The class is annotated with `@RunWith(CamelCdiRunner.class)` **1**, enabling running the test with Camel and CDI. `CamelCdiRunner` is provided by the `camel-test-cdi` component. The test uses `FluentProducerTemplate` **2** to send a test message to the Camel hello service **3**. Notice that the template is injected using both `@Inject` and `@Uri` **2**. The endpoint URI defined in `@Uri` represents the default endpoint, which is optional. The unit test could have been written as follows:

```
@Inject @Uri ①
```

①

Injects template without a default endpoint URI

```
private FluentProducerTemplate producer;  
  
@Test  
public void testHello() throws Exception {  
    String out =  
producer.to("jetty:http://localhost:8080/hello") ②
```

②

Specifies the URI of the endpoint to send the message to

```
        .request(String.class);  
    assertTrue(out.startsWith("Hello from Camel CDI"));  
}
```

FluentProducerTemplate isn't configured with an endpoint URI ①, which we then must provide when we send the message ②.

This example is provided with the accompanying source code in the chapter7/cdi directory.

Before reaching the end of our mini coverage of using Camel CDI, we'll show you one last feature of CDI that you can use with Camel: event listening.

LISTENING TO CAMEL EVENTS USING CDI

CDI supports an event notification mechanism that allows you to listen for certain events and react to them. This can be used to listen to Camel lifecycle events such as when Camel Context or routes start or stop. The following code shows how to listen for Camel starting:

```
@Singleton  
public class HelloConfiguration {  
  
    void onContextStarted(@Observes CamelContextStartedEvent  
event) { ①
```

1

Listens for CamelContextStartedEvent to happen

```
System.out.println("*****");
System.out.println("* Camel started " +
event.getContext().getName());
System.out.println("*****");
}
}
```

To listen for events in CDI, you declare a method with a parameter that carries the event class to listen for. The parameter must be annotated with the @Observes annotation ①.

This concludes our coverage of using Camel with CDI, including coverage of three of the most common, key features of CDI and camel-cdi:

- Dependency injection using @Inject and @Produces
- Injecting a Camel endpoint using @Uri
- Bean lifecycle using scopes such as @Singleton
- Event listening using @Observes

Before moving on to the next Camel microservice runtime, we want to share some of our thoughts on using Camel CDI.

SUMMARY OF USING CAMEL CDI FOR MICROSERVICES

Developers wanting to build microservices by using Java code can find value in using a dependency injection framework such as CDI or Spring Boot. Camel users who find the XML DSL attractive shouldn't despair because camel-cdi supports both Java and XML DSLs.

TIP You can find more documentation about using camel with CDI at <http://camel.apache.org/cdi>.

What CDI brings to the table is a Java development model using Java code and annotations to configure Java beans and specify their interrelationships.

When using CDI, you need a CDI runtime such as JBoss Weld, which we've been using in our examples. JBoss Weld isn't primarily intended as a standalone server but finds its primary use-case as a component inside an existing application server such as WildFly, WildFly Swarm, or Apache TomEE. Attempting to build your application as a fat JAR deployment and run with JBoss Weld isn't as easy. Instead, you should look at a more powerful application server such as WildFly Swarm or Spring Boot, which are covered next.

7.2.3 WILDFLY SWARM WITH CAMEL AS MICROSERVICE

WildFly Swarm is a Java EE application server in which you package your application and the bits from the WildFly Swarm server you need together in a *fat JAR* binary. You can also view WildFly Swarm as Spring Boot but for Java EE applications.

It's easy to get started with WildFly Swarm in a new project, as you set up your Maven pom.xml by doing the following:

- Import WildFly Swarm bill of materials (BOM) dependency
- Declare the WildFly Swarm dependencies you need (such as Camel, CDI, and so forth)
- Add the WildFly Swarm Maven plugin that generates the fat JAR

You can easily get started with a new project by using the generator web page shown in [figure 7.1](#).

In the previous section, you built a Camel microservice using CDI. Now you want to take that application and run it on

WildFly Swarm, so you choose Camel CDI and CDI as dependencies. You then need to make one other change, because you're using Jetty as the HTTP server, but WildFly Swarm comes out of the box with its own HTTP server named Undertow. This requires you to change the source code, as shown in Listing 7.5.

The screenshot shows the WildFly Swarm Project Generator web interface. At the top, there's a navigation bar with links for GENERATOR, BLOG, DOCUMENTATION, COMMUNITY, and RESOURCES. Below the navigation, the title 'WildFly Swarm Project Generator' is displayed, followed by the subtext 'Rightsize your Java EE microservice in a few clicks'. A 'Instructions' section contains a numbered list of steps: 1. Choose the dependencies you need, 2. Click on the Generate button to download the hello.zip file, 3. Unzip the file in a directory of your choice, and 4. Run mvn wildfly-swarm:run in the unzipped directory. Below these instructions, there are fields for 'Group ID' (containing 'camelinaction'), 'Artifact ID' (containing 'hello'), and a 'Generate Project' button. Under 'Dependencies', there's a search bar with the placeholder 'JAX-RS, EJB, Transactions, Ribbon, Hibernate Search...' and a link 'Not sure what you are looking for? View all available dependencies filtered by: All'. Below the search bar, there's a 'Selected dependencies' section with two items: 'Camel Component :: Cdi' and 'CDI'.

Figure 7.1 Creating a new WildFly Swarm project from the generator web page (<http://wildfly-swarm.io/generator/>). Here we've chosen Camel CDI and CDI as the dependencies we need. Clicking the view all dependencies link shows all the dependencies you can choose from. Clicking Generate Project downloads a zip file with the generated source code.

Listing 7.5 Hello service using Undertow as HTTP server

```
@Singleton  
public class HelloRoute extends RouteBuilder {
```

```
@Inject  
private HelloBean hello;  
  
@Inject @Uri("undertow:http://localhost:8080/hello")
```

Use Undertow as HTTP server as it comes out of the box with WildFly Swarm

```
private Endpoint undertow;  
  
@Override  
public void configure() throws Exception {  
    from(undertow)  
        .bean(hello, "sayHello");  
}  
}
```

And you also need to add camel-undertow as a dependency to your Maven pom.xml file, as shown here:

```
<dependency>  
    <groupId>org.wildfly.swarm</groupId>  
    <artifactId>camel-undertow</artifactId>  
</dependency>
```

Notice that the groupId of the dependency isn't org.apache.camel, but is org.wildfly.swarm. The reason is that WildFly Swarm provides numerous supported and curated dependencies. These dependencies are called *fragments* in WildFly Swarm. In this case, there's a fragment for making Camel and Undertow work together, hence you must use this dependency.

After this change, the service is ready to run on WildFly Swarm, which you can try from the source code by running the following Maven goal from the chapter7/wildfly-swarm directory:

```
mvn wildfly-swarm:run
```

Then from a web browser you can call the service from the following URL: http://localhost:8080/hello.

You can also run the example as a fat JAR by executing the following:

```
java -jar target/hello-swarm.jar
```

This was a brief overview of using Camel with WildFly Swarm. We covered most parts in the previous section about using CDI with Camel.

One important aspect to know when using WildFly Swarm and Camel is the importance of using fragments over Camel components.

WILDFLY SWARM FRAGMENTS VERSUS CAMEL COMPONENTS

When using the Java EE functionality from WildFly Swarm with Camel, you should use the provided WildFly-curated components, called *fragments*. The example uses Undertow, and therefore should use the WildFly Swarm fragment of camel-undertow, and not the regular camel-undertow component:

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>camel-undertow</artifactId>
</dependency>
```

When there's no specialized fragment for a Camel component, you should use the regular component. For example, there's no fragment for the stream component, and as a user you should use camel-stream:

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

You can see a list of all known Camel fragments from the WildFly Swarm generator web page by clicking View All Available Dependencies (as was shown in [figure 7.1](#)).

CDI isn't the only option when using Camel on WildFly Swarm. You can also use Camel routes defined in Spring XML

files.

USING CAMEL ROUTES WITH SPRING XML

Many Camel users are using the Camel XML DSL to define routes in XML in either Spring or OSGi Blueprint files. Spring is supported by WildFly Swarm, so you can use Spring XML files to define Camel routes and beans. The route in [listing 7.6](#) could be done in Spring XML:

```
<bean id="helloBean" class="camelinaction.HelloBean"/>

<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <propertyPlaceholder id="properties"
        location="classpath:hello.properties"/>
        <route>
            <from uri="undertow:http://localhost:8080/hello"/>
            <bean ref="helloBean" method="sayHello"/>
        </route>
</camelContext>
```

Spring XML files must be stored in the `src/main/resources/spring` directory, and the name of the files must use the suffix `-camel-context.xml`. The book's source code includes this example in the `chapter7/wildfly-swarm-spring` directory, which you can try by using the following Maven goal:

```
mvn wildfly-swarm:run
```

MONITORING CAMEL WITH WILDFLY SWARM

WildFly Swarm comes with monitoring and health checks out of the box provided by the `monitor` fraction. To enable monitoring, you need to add the `monitor` fraction to the Maven `pom.xml` file:

```
<dependency>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>monitor</artifactId>
</dependency>
```

You can try this by running the `chapter7/wildfly-swarm-spring`

example:

```
mvn wildfly-swarm:run
```

Then, from a web browser, open `http://localhost:8080/node`, which shows overall status of the running WildFly Swarm node. Another endpoint is `/health`, which shows health information. At the time of this writing, there are no default health checks installed and the endpoint returns an HTTP Status 204:

```
$ curl -i http://localhost:8080/health
HTTP/1.1 204 No health endpoints configured!
```

But it's expected that the Camel fraction will include a Camel-specific health check in a future release.

We'll end our coverage of WildFly Swarm by sharing our thoughts of the good and bad when using Camel with WildFly Swarm.

SUMMARY OF USING WILDFLY SWARM WITH CAMEL FOR MICROSERVICES

WildFly Swarm is a lightweight, *just-enough* application server that provides support for all of the Java EE stack. This is appealing for users who are already using Java EE. Users not using Java EE can also use WildFly Swarm and find value in using standards such as CDI, JAX-RS, and others. Camel users preferring the XML DSL can use WildFly Swarm with the Spring XML supported out of the box.

Because WildFly Swarm does independent releases of the WildFly Camel fragments, you may find yourself using a newer version of Apache Camel that hasn't had a WildFly Swarm Camel release yet, meaning you're stuck using the older version.

WildFly Swarm isn't the only just-enough server on the market. Spring Boot is another popular choice that works well with Camel.

7.2.4 SPRING BOOT WITH CAMEL AS MICROSERVICE

Spring Boot is an opinionated framework for building microservices. Spring Boot is designed with convention over configuration and allows developers to get started quickly developing microservices with reduced boilerplate, configuration, and fuss. Spring Boot does this via the following:

- Autoconfiguration and reduced configuration needed
- Curated list of starter dependencies
- Simplified application packaging as a standalone fat JAR
- Optional application information, insights, and metrics

Let's get started with Spring Boot. First we'll try without Camel, and then we'll add Camel to an existing Spring Boot application.

GETTING STARTED WITH SPRING BOOT

We'll be using the Spring Boot starter website (<http://start.spring.io>) to get started. As dependencies, we chose only Web in order to start with a plain web application. You do this by typing `web` in the Search for dependencies field and then selecting Web in the list presented. [Figure 7.2](#) shows where we're going.

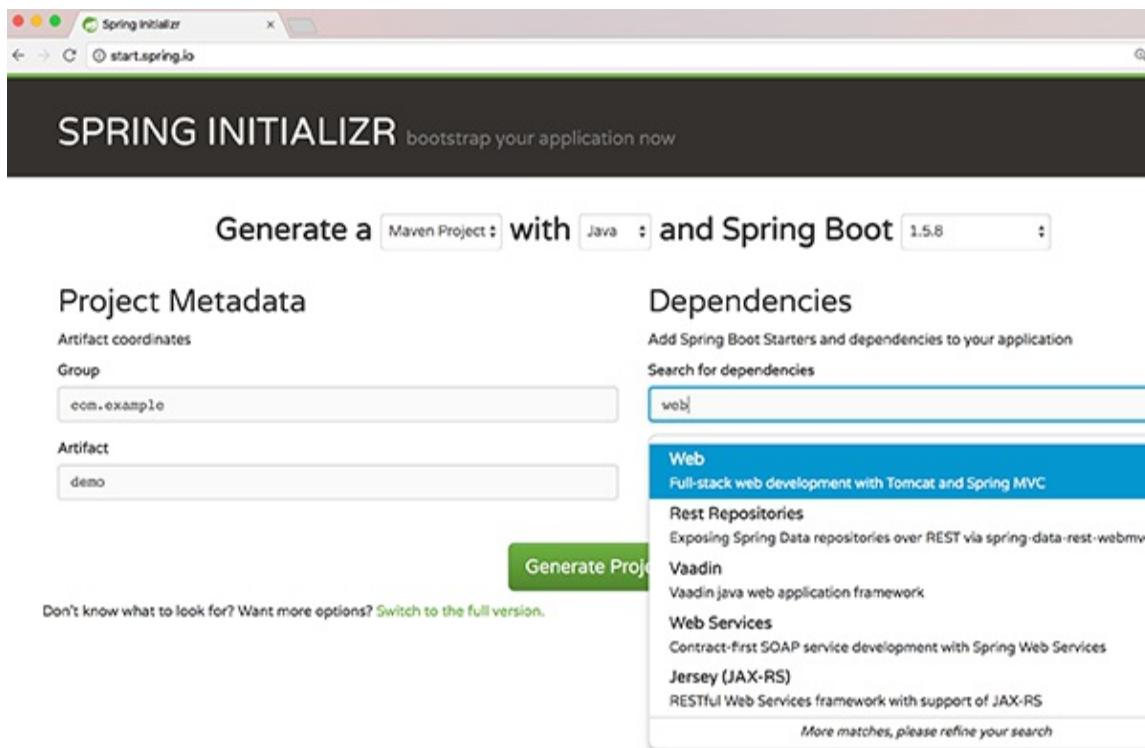


Figure 7.2 Create a new Spring Boot project from the Spring starter website. Start typing `web` in the Search for Dependencies field and then select Web in the list to add it as dependency.

You click the Generate Project button to download the project. You then unzip the downloaded source code and are ready to run the Spring Boot application by using Maven:

```
mvn spring-boot:run
```

TIP You can also create a new Spring Boot project using Spring CLI, or the camel-archetype-spring-boot Maven archetype from Apache Camel. You'll learn how to use the Camel Maven archetypes in chapter 8.

The application then starts up, and you can navigate to `http://localhost:8080` in your browser and see a web page. Our application doesn't do anything yet, so let's add a REST endpoint to return a hello message.

ADDING REST TO SPRING BOOT APPLICATION

You want to add a REST endpoint that returns a simple hello message. Spring provides REST support out of the box, which you can build using Java code, as shown in the following listing.

Listing 7.6 `RestController` exposes a REST endpoint using Spring Rest

```
@RestController
```

1

Define this class as a REST endpoint

```
@RequestMapping("/spring")
```

2

Root mapping for all requests

```
public class HelloRestController {  
  
    @RequestMapping(method = RequestMethod.GET, value =  
    "/hello",  
        produces = "text/plain")
```

3

HTTP GET service mapped to /hello

```
    public String hello() {  
        return "Hello from Spring Boot";  
    }  
}
```

The class `HelloRestController` is annotated with `@RestController` 1, which tells Spring that this class is a REST controller exposing REST endpoints. The annotation `@RequestMapping` is used to map the HTTP URI to Java classes, methods, and parameters. For example, the annotation 2 at the class level maps all HTTP requests starting with `/spring` in the context path

to this controller. The annotation ❸ at the method exposes a REST service under /spring/hello as a HTTP GET service that produces plain-text content.

The book's source code carries this example in the chapter7/springboot directory; you can try the example by using the following Maven goal:

```
mvn spring-boot:run
```

Then from a web browser, open
<http://localhost:8080/spring/hello>.

In the sections to follow, we'll show you two ways of adding Camel to Spring Boot. We'll add Camel to the existing HelloRestController class and then add Camel routes.

ADDING CAMEL TO EXISTING SPRING BOOT REST ENDPOINT

Suppose you have an existing Spring REST controller and want to use one of the many Camel components. You can easily add Camel to any existing Spring controller classes by dependency-injecting a Camel ProducerTemplate or FluentProducerTemplate and then using the template from the controller methods, as shown in the following listing.

Listing 7.7 Adding Camel to Spring controller class

```
@RestController
@RequestMapping("/spring")
public class HelloRestController {

    @EndpointInject(uri = "geocoder:address:current") ❶
```

❶

Dependency-injecting Camel FluentProducerTemplate

```
    private FluentProducerTemplate producer;

    @RequestMapping(method = RequestMethod.GET, value =
"/hello",
```

```
        produces = "text/plain")
public String hello() {
    String where =
producer.request(String.class); ②
```

②

Using the template to request the endpoint and receive the response as a String

```
        return "Hello from Spring Boot and Camel. We are
at: " + where;
}
}
```

One of the best ways to use Camel from an existing Java class such as a Spring controller is to inject a Camel `ProducerTemplate` or `FluentProducerTemplate` **①**, where you can define the endpoint to call. In this example, we want to know the current location where the application runs by using the `camel-geocoder` component. This component contacts a service on the internet that, based on your IP, can track your location (to some degree—one of our current locations was 14 kilometers off by the geocoder). The controller then uses the injected Camel template to contact the geocoder when the `hello` method is invoked **②** so the location can be included in the response.

TIP If you want to use a Camel route from the controller class, you can use the `direct` component to call the Camel route.

USING CAMEL STARTER COMPONENTS WITH SPRING BOOT

When using Spring Boot with Camel, you should favor using the Camel starter components that are curated to work with Spring Boot. This example uses the following Camel components in the Maven `pom.xml` file:

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

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

Notice that the `artifactId` of the Camel components uses the suffix `-starter`. For every Camel component, there's a corresponding Camel Spring Boot-curated starter component that's recommended to use. The starter component has been specially adjusted to work with Spring Boot and includes support for Spring Boot autoconfiguration, which we'll cover later in this section.

The book's source code contains this example in `chapter7/springboot-rest-camel`, and you can try it by using the following goal:

```
mvn spring-boot:run
```

Then from a web browser, open `http://localhost:8080/spring/hello`.

This example doesn't use any Camel routes. Let's take a look at building this example by using a Camel route instead of a Spring controller.

USING CAMEL ROUTES WITH SPRING BOOT

Camel routes are natural to Camel applications, and they're supported in Spring Boot. You can write your Camel routes in Java or XML DSL, but because Spring Boot is Java-centric, you may take on using Java DSL instead of XML. Let's rewrite the previous example that uses a Spring controller to expose a REST service to use a Camel route instead. The following listing shows how this can be done.

Listing 7.8 Use Camel route in Spring Boot to expose a REST service

1

@Component annotation to let Camel route be autodiscovered

```
public class HelloRoute extends RouteBuilder {  
  
    @Override  
    public void configure() throws Exception {  
        rest("/").produces("text/plain") ②  

```

2

Camel Rest DSL to define a REST service in the route

```
        .get("hello")  
        .to("direct:hello");  
  
    from("direct:hello") ③
```

3

Camel route

```
        .to("geocoder:address:current")  
        .transform().simple("Hello. We are at:  
${body}");  
    }  
}
```

When creating Camel routes in Java DSL with Spring Boot, you should annotate the class with `@Component`, which ensures that the route is autodetected by Spring and automatically added to Camel when the application starts up ①. Because we want to expose a rest service from a Camel route, we need to use an HTTP server component. Spring Boot comes with a servlet engine out of the box, therefore we need to configure a servlet to be used. You can do this easily with Camel by adding the `camel-servlet-starter` dependency to the Maven `pom.xml` file. Camel will then automatically detect the `camelServlet` and integrate that with the servlet engine within Spring Boot. Camel will by default use the context-path `/camel/*`, which can be reconfigured in the `application.properties` file.

In Camel, you can define REST services using a DSL known as Rest DSL ②, which will use the servlet component that's just been configured. The Camel route ③ comes next. Notice that the Rest DSL calls the route by using the direct endpoint, which is how you can link them together.

NOTE Chapter 10 covers the Rest DSL extensively. But we thought the three lines of Rest DSL code shown in [listing 7.8](#) wouldn't knock you down.

We've provided the source code for this example in the chapter7/springboot-camel directory; you can try it by using the following Maven goal:

```
mvn spring-boot:run
```

Then from a web browser, open <http://localhost:8080/camel/hello>.

Some Camel users prefer using XML DSL over Java DSL.

USING CAMEL XML DSL WITH SPRING BOOT

Spring Boot supports loading Spring XML files, which is how using the XML DSL would work. In your Spring Boot application, you use the `@ImportResource` annotation to specify the location of the XML file from the classpath, as shown here:

```
@SpringBootApplication  
@ImportResource("classpath:mycamel.xml")  
public class SpringbootApplication {
```

The XML file is a regular Spring `<beans>` XML that can contain `<bean>`s and `<camelContext>`, as shown here:

```
<beans ...>  
  <camelContext  
  xmlns="http://camel.apache.org/schema/spring">  
    <route>  
      <from uri="timer:foo"/>
```

```
        <log message="Spring Boot says {{hello}} to me"/>
    </route>
</camelContext>
</beans>
```

Being able to load Spring XML files in your Spring Boot applications is useful for users who either prefer Camel XML DSL or are migrating existing Spring applications that have been configured with Spring XML files to run on Spring Boot.

We've provided an example with the source code in chapter7/springboot-xml that you can try by using the following Maven goal:

```
mvn spring-boot:run
```

The perceptive eye may notice that this example uses a property placeholder in the Spring XML file in the <log> EIP.

USING CAMEL PROPERTY PLACEHOLDERS WITH SPRING BOOT

Spring Boot supports property configuration out of the box, which allows you to specify your placeholders in the application.properties file. Camel ties directly into Spring Boot, which means you can define placeholders in the application.properties files that Camel can use. The previous example uses the Camel property placeholder in the <log> EIP:

```
<log message="Spring Boot says {{hello}} to me"/>
```

We then define a property with the key hello in application.properties:

```
hello=I was here
```

And hey, presto, no surprise: Camel is able to look up and use that value without you having to configure anything.

Spring Boot allows you to override property values in various ways. One of them is by configuring JVM system properties. Instead of using `hello=I was here`, you can override this when you start the JVM as follows:

```
mvn spring-boot:run -Dhello='Donald Duck'
```

Another possibility is to use OS environmental variables:

```
export HELLO="Goofy"  
mvn spring-boot:run
```

You can also specify the environment variable as a single command line:

```
HELLO=Goofy mvn spring-boot:run
```

Where are we going with this? This allows Spring Boot applications to easily externalize configuration of your applications.

In section 7.1.1, we talked about the characteristics of a microservice, with one of them being that they're highly configurable. Another characteristic of a microservice is that it should be observable by monitoring and management systems.

MONITORING CAMEL WITH SPRING BOOT

Spring Boot provides *actuator endpoints* to monitor and interact with your application. Spring Boot includes built-in endpoints that can report many kinds of information about the application.

To enable Spring Boot actuator, you'd need to add its dependency to the Maven pom.xml file as shown here:

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-actuator</artifactId>  
</dependency>
```

One of these endpoints is the health endpoint, which shows application health. Spring Boot allows third-party libraries to provide custom health indicators, which Camel has. This means your Camel applications can provide health indicators whether the Camel application is healthy or not.

You can try this by running the chapter7/springboot-rest-camel example:

```
mvn spring-boot:run
```

Then from a web browser, open: <http://localhost:8080/health>.

By always using the /health endpoint for your Spring Boot applications, you can more easily set up centralized monitoring to use this endpoint to query the health information about your running applications.

Next we have a few words to say about why Camel and Spring Boot work well together for developing microservices applications.

SUMMARY OF USING CAMEL WITH SPRING BOOT FOR MICROSERVICES

Spring Boot is a popular *just-enough* application runtime for running your Java applications. Camel has always had first-class support for Spring, and using Camel with Spring Boot is a great combination. We think it's a great choice. But if you're using or coming from a Java EE background, we recommend looking at WildFly Swarm instead.

No man is an island is a famous phrase from John Donne's 400-year-old poem. Likewise, no microservice is alone. Microservices are composed together to conduct business transactions. The next topic is about building microservices with Camel that call other services and the implications that brings to the table.

7.3 *Calling other microservices*

In the land of microservices, each service is responsible for providing functionality to other collaborators. Two of the microservices' characteristics we discussed at the beginning of this chapter are that building distributed systems is hard and that microservices must be designed to deal with failures. This section first covers the basics: how to build Camel-based microservices that call other services (without design for failures). The subsequent section covers the EIP patterns that

can be applied to design for failures.

THE STORY OF THE BUSY DEVELOPER

At Rider Auto Parts, you've been dazzling a bit with microservices but haven't kicked the tires yet because of a busy work schedule. Does it all sound too familiar? Yeah, even authors of Camel books know this feeling too well. To put yourself back in the saddle again, you send your spouse and kids away to visit your in-laws, leaving you alone in the house for the entire weekend. Your goal is to enjoy an occasional beer while studying and writing some code to familiarize yourself with building a set of microservices that collectively solve a business case. You don't want to settle on a particular runtime and therefore want to implement the services using various technologies.

The business case you'll attempt to implement by using microservices is illustrated in [figure 7.3](#).

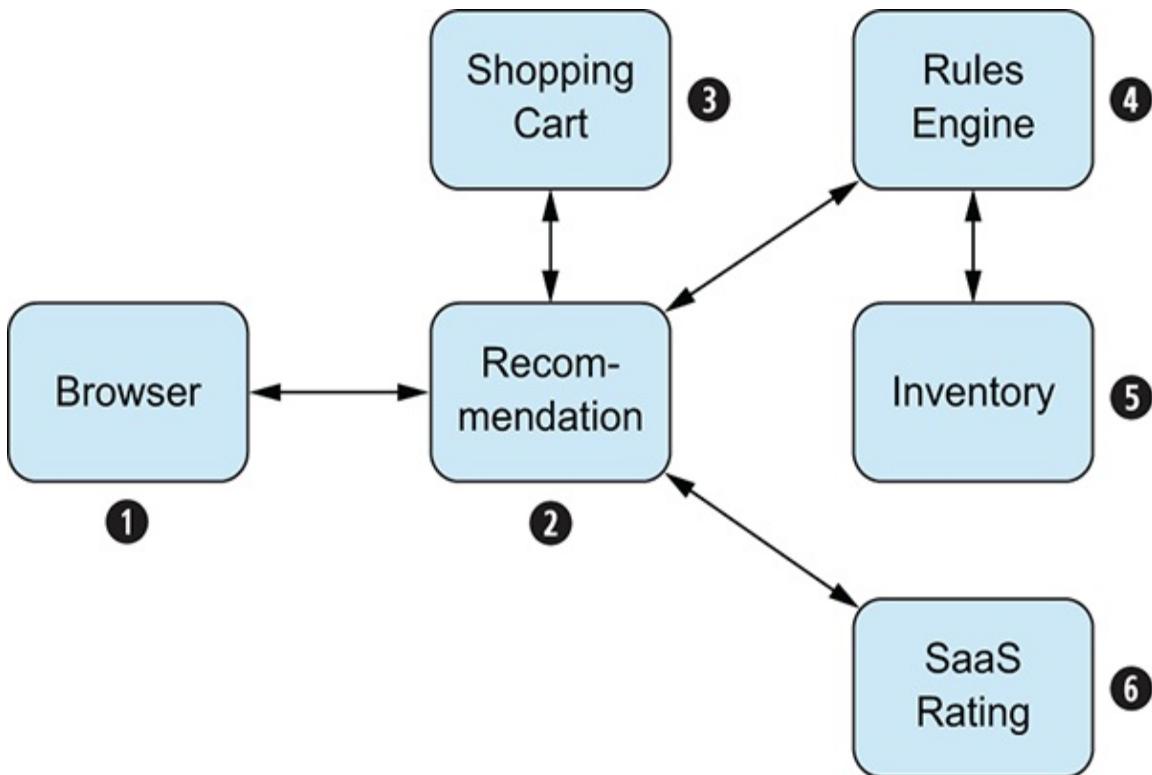


Figure 7.3 Rider Auto Parts recommendation system. From the web browser ①, the recommendation service ② obtains items currently in the shopping cart ③ and then calls the rules engine ④ to compute items to be recommended, which

takes into account the number of items currently in stock **⑤**. Finally, an external user-based ranking system from a cloud provider **⑥** is used to help rank recommendations.

Rider Auto Parts has an upcoming plan to implement a new recommendation system, and you decide to get a jump start by implementing a prototype. Users who are browsing the Rider Auto Parts website will, based on their actions, have recommended items displayed. The web browser **①** communicates with the recommendation microservice **②**. The recommendation service calls three other microservices **③ ④ ⑥** as part of computing the result. Items already in the shopping cart **③** are fed into the rules engine **④**, which is used to rank recommendations. The inventory system **⑤** in the back end hosts information about the items currently in stock and can help rank items that can be shipped to customers immediately. An external SaaS service is used to help rank popular items based on a worldwide user-based rating system **⑥**.

TECHNOLOGIES

The technology stack for the recommendation system is illustrated in figure 7.4.

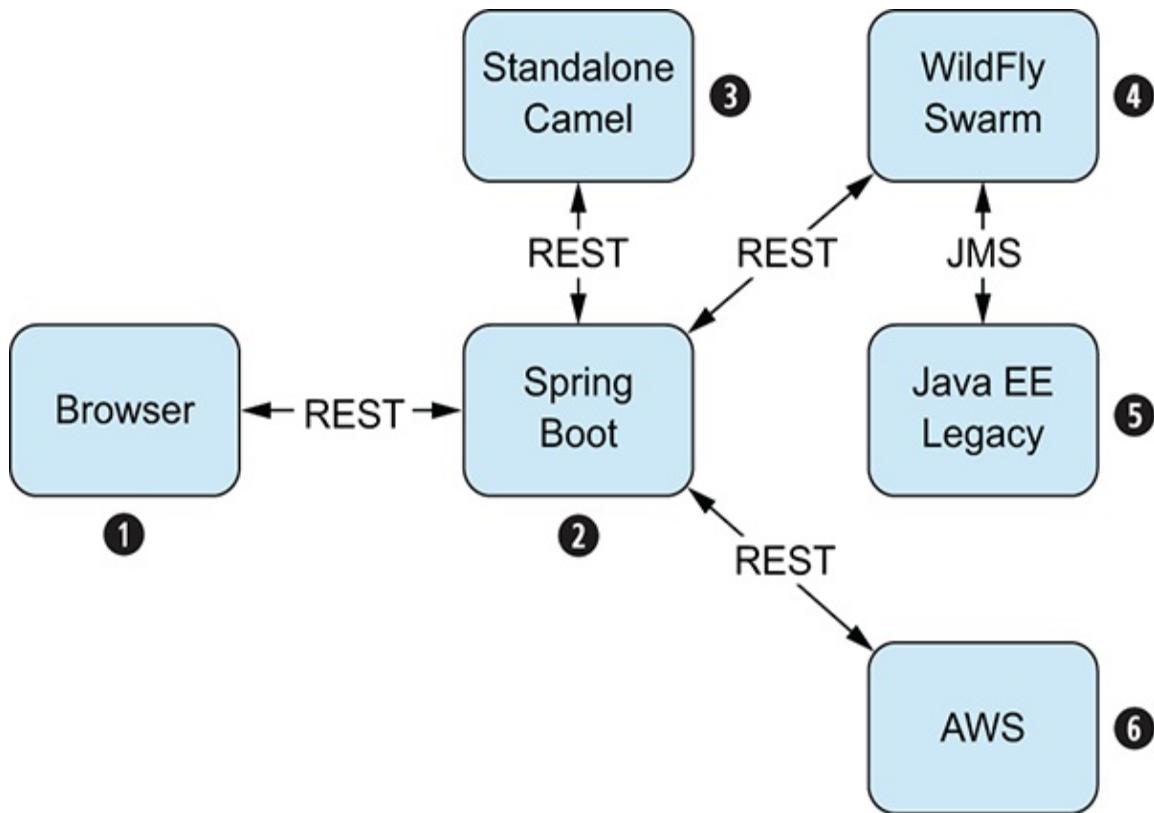


Figure 7.4 Web browser ① communicates using HTTP/REST to the recommendation system ② running on Spring Boot. The shopping cart is running standalone Camel ③. The rules engine ④ running inside WildFly Swarm is integrated using REST. The inventory system is hosted on a legacy Java EE application server ⑤, which is integrated using JMS messaging. The rating system is hosted in the cloud on AWS ⑥.

You have one weekend, so let's start hacking code. You've decided to implement each microservice as a separate module in the source code. To keep things simple, you decide to keep all the source in the same Git repository. The source code structure is like this:

<pre>prototype ├── cart └── inventory └── recommend └── rules └── rating</pre>	<pre>Shopping cart Inventory system Recommendation microservice Rules engine External SaaS rating system</pre>
--	--

You can find the source code for this example in the chapter7/prototype directory. The following section covers how to build this prototype and highlights the key parts.

7.3.1 RECOMMENDATION PROTOTYPE

The prototype for the recommendation microservice is built as a vanilla Spring Boot application. This microservice is the main service, which depends on all the other microservices, and hence has more moving parts than the services to follow.

The prototype consists of the following source files:

- *CartDto.java*—POJO representing a shopping cart item
- *ItemDto.java*—POJO representing an item to be recommended
- *RatingDto.java*—POJO representing a rating of an item
- *RecommendController.java*—The REST service implementation
- *SpringBootApplication.java*—The main class to bootstrap this service
- *application.properties*—Configuration file

The *meat* of this service is the `RecommendController` class, shown in the following listing.

Listing 7.9 Recommendation REST service

```
@RestController
```

①

①

Defines class as REST controller

```
@RequestMapping("/api")
@ConfigurationProperties(prefix = "recommend")
```

②

②

Injects configuration properties with prefix “recommend”

```
public class RecommendController {
```

```
    private String cartUrl;
```

③

3

Fields with injected configuration values

```
private String rulesUrl;    3  
private String ratingsUrl; 3  
  
private final RestTemplate restTemplate = new  
RestTemplate(); 4
```

4

REST template used to call other microservices

```
@RequestMapping(value = "recommend", method =  
RequestMethod.GET,  
produces = "application/json") 5
```

5

Defines REST service

```
public List<ItemDto> recommend(HttpSession session) {  
    String id = session.getId();  
  
    CartDto[] carts = restTemplate.getForObject(  
        cartUrl, CartDto[].class,  
        id); 6
```

6

Calls shopping cart REST service

```
String cartIds = cartsToCommaString(carts);  
  
ItemDto[] items = restTemplate.getForObject(  
    rulesUrl, ItemDto[].class, id,  
    cartIds); 7
```

7

Calls rules engine REST service

```
String itemIds = itemsToCommaString(items);
```

```
RatingDto[] ratings = restTemplate.getForObject(  
    ratingsUrl, RatingDto[].class,  
    itemIds); 8
```

8

Calls SaaS rating REST service

```
for (RatingDto rating : ratings) {  
    appendRatingToItem(rating, items);  
}  
return Arrays.asList(items);  
}
```

Spring Boot makes it easy to define the REST service in Java by annotating the class with `@RestController` and `@RequestMapping` **1**. The chapter introduction indicated the good practice of externalizing your configuration. Spring Boot allows you to automatically inject getter/setters **3** from its configuration file by annotating the class with `@ConfigurationProperties` **2**. In this example, we've configured `recommend` as the prefix that maps to the following properties from the `application.properties` file:

```
recommend.cartUrl=http://localhost:8282/api/cart?sessionId=  
{id}  
recommend.rulesUrl=http://localhost:8181/api/rules/{cartIds}  
recommend.ratingsUrl=http://localhost:8383/api/ratings/{ite  
mIds}
```

To call other REST services, we use Spring's `RestTemplate` **4**. The method `recommend` is exposed as a REST service by the `@RequestMapping` annotation **5**. The service then calls three other microservices **6** **7** **8** using `RestTemplate`.

Automatic mapping JSON response to DTO

Notice that the `RestTemplate` `getForObject` method automatically maps the returned JSON response to an array of the desired DTO classes, such as when calling the shopping cart service:

```
CartDto[] carts = restTemplate.getForObject(cartUrl,  
                                         CartDto[].class, id);
```

You must use an array type and can't use a `List` type because of Java type erasure in collections. For example, the following can't compile:

```
List<CartDto> carts =  
restTemplate.getForObject(cartUrl,  
                         List<CartDto.class>, id);
```

As you can see, calling another microservice using Spring's `RestTemplate` is easy, because it requires just one line of code with `RestTemplate`. Have you seen this kind before? Yes: Camel's `ProducerTemplate` or `FluentProducerTemplate` is also easy to use for sending messages to Camel endpoints with one line of code.

In [listing 7.10](#), the first microservices called is the shopping cart service.

7.3.2 SHOPPING CART PROTOTYPE

A goal of building the prototype is also to gain practical experience by using Camel with different Java toolkits and frameworks. This time, you've chosen to use standalone Java with CDI and Camel for the shopping cart service. The implementation of the shopping cart is kept simple as a pure in-memory storage of items currently in the carts, as shown in the following listing.

Listing 7.10 Simple shopping cart implementation class

`@ApplicationScoped`

1

1

CDI application-scoped bean named cart

```
@Named("cart")
public class CartService {

    private final Map<String, Set<CartDto>> content = new
LinkedHashMap<>();

    public void addItem(@Header("sessionId") String
sessionId,
                        @Body CartDto dto) { 2
```

2

Method to add item to cart stored

```
Set<CartDto> dtos = content.get(sessionId);
if (dtos == null) {
    dtos = new LinkedHashSet<>();
    content.put(sessionId, dtos);
}
dtos.add(dto);
}

public void removeItem(@Header("sessionId") String
sessionId,
                      @Header("itemId") String itemId)
{ 3
```

3

Method to remove an item from the cart

```
Set<CartDto> dtos = content.get(sessionId);
if (dtos != null) {
    dtos.remove(itemId);
}
}

public Set<CartDto> getItems(@Header("sessionId") String
sessionId) { 4
```

4

Method to get the items from the cart

```

        Set<CartDto> answer = content.get(sessionId);
        if (answer == null) {
            answer = Collections.EMPTY_SET;
        }
        return answer;
    }
}

```

The `CartService` class has been annotated with `@ApplicationScoped` ❶ because only one instance is needed at runtime. The name of the instance can be assigned by using `@Named`. The class implements three methods ❷ ❸ ❹ to add, remove, and get items from the cart. Notice that these methods have been annotated with Camel's `@Header` to bind the parameters to message headers. We do this to make it easy to call the `CartService` bean from a Camel route, which we'll cover in a little while.

TIP We covered bean parameter bindings in chapter 4.

In [listing 7.10](#), the shopping cart uses `CartDto` in the `add` and `get` methods ❷ ❹. This class is implemented as a plain POJO, as shown here:

```

public class CartDto {
    private String itemId;
    private int quantity;

    @ApiModelProperty(value = "Id of the item in the shopping
cart") ❶

```

❶

Swagger annotation to document the fields

```

public String getItemId() {
    return itemId;
}

@ApiModelProperty(value = "How many items to
purchase") ❶

```

```
public int getQuantity() {  
    return quantity;  
}  
  
// setter methods omitted  
}
```

The `CartDto` class has two fields to store the item ID and the number of items to purchase. The getter methods ❶ have been annotated with Swagger's `ApiModelProperty` to include descriptions of the fields, which will be used in the API documentation of the microservice.

TIP Chapter 10 covers REST services and Swagger API documentation.

Earlier in this chapter, in code [listing 7.5](#), you got a quick glimpse of Camel's Rest DSL. Now brace yourself, because this time we'll dive deeper as you implement the shopping cart microservice using Camel's Rest DSL and with automatic API documentation using Swagger. All this is done from the Camel route shown in the following listing.

[Listing 7.11](#) Shopping cart microservice using Camel's Rest DSL

```
@Singleton  
public class CartRoute extends RouteBuilder {  
  
    public void configure() throws Exception {  
  
        restConfiguration("jetty").port="{{port}}") ❶
```

❶

Configures Rest DSL to use Jetty component

```
.contextPath("api") ❶  
.bindingMode(RestBindingMode.json) ❷
```

2

Turns on JSON binding to/from POJO classes

```
.dataFormatProperty("disableFeatures",  
"FAIL_ON_EMPTY_BEANS") ③
```

3

Turns off binding error on empty lists/beans

```
.apiContextPath("api-doc") ④
```

4

Enables Swagger API documentation

```
.enableCORS(true);  
  
rest("/cart").consumes("application/json").produces("application/json")  
    .get() ⑤
```

5

GET/cart service

```
.outType(CartDto[].class)  
    .description("Returns the items currently in the  
shopping cart")  
    .to("bean:cart?method=getItems") ⑧
```

8

Calls the CartService bean methods

```
.post() ⑥
```

6

POST /cart service

```
.type(CartDto.class)
```

```
.description("Adds the item to the shopping cart")
.to("bean:cart?method=addItem") 8
7
.delete().description("Removes the item from the
shopping cart") 7
```

7

DELETE /cart service

```
.param().name("itemId")
.description("Id of item to remove")
.endParam()
.to("bean:cart?method=removeItem"); 8
}
}
```

The REST service is using Camel's jetty component **1** as the HTTP server. To work with JSON from Java POJO classes, we turned on JSON binding **2**, which will use Jackson under the covers. Jackson is instructed to not fail if binding from an empty list/bean **3**. This is needed in case the shopping cart is empty and the GET service **6** is called. Swagger is turned on for API documentation **4**. The Rest DSL then exposes three REST services **5** **6** **7**, each calling the cartService bean **8**. Notice that each REST service documents its input and output types and parameters, which becomes part of the Swagger API documentation.

RUNNING THE SHOPPING CART SERVICE

You can start the source code for the shopping cart microservice, located in the chapter7/prototype/cart directory, using the following Maven goal:

```
mvn compile exec:java
```

You can then access the Swagger API documentation from a web browser:

```
http://localhost:8282/api/api-doc
```

To test the shopping cart, you can install Postman, which has a

built-in REST client, as an extension to your web browser, as shown in [figure 7.5](#).

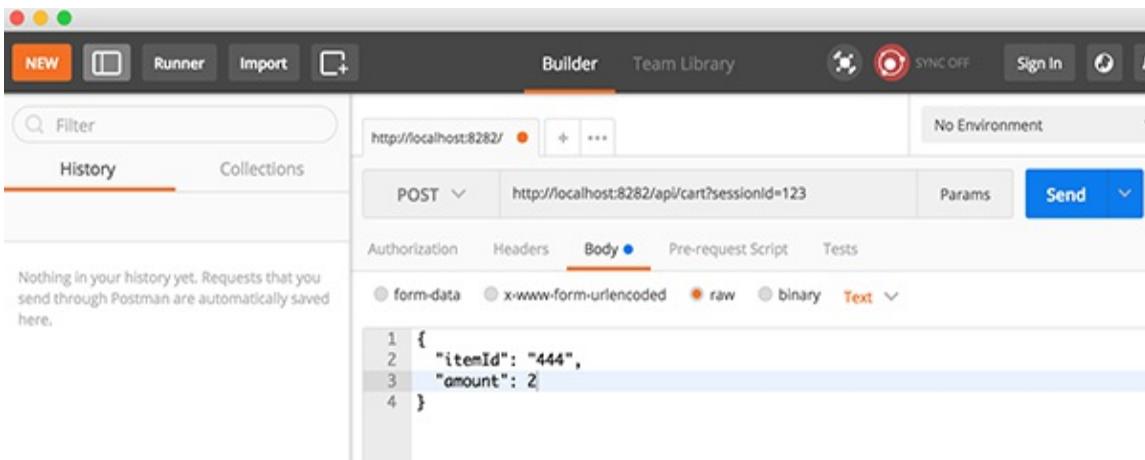


Figure 7.5 Using Postman to send an HTTP POST request to a shopping cart service to add the item to the cart

Using Postman, you can then more easily call REST services with more-complex queries that involve `POST`, `PUT`, and `DELETE` operations. Notice that we hardcode the HTTP session ID to `123` as a query parameter.

Because the shopping cart microservice includes Swagger API documentation, we can use Swagger UI to test the service. You can easily run Swagger UI using Docker by running the following command line:

```
docker run -d --name swagger-ui -p 8080:8080
swaggerapi/swagger-ui
```

Open a web browser:

```
http://localhost:8080/
```

Then type the following URL in the URL field and click the Explore button:

```
http://localhost:8282/api/api-doc
```

You should see the cart service, which you can expand and try, as shown in [figure 7.6](#).

This was a glimpse of some of the powers Camel provides for

REST services. We've devoted all of chapter 10 to covering this in much more depth. Let's move on to the next microservice you're developing during the weekend: the rules and inventory services.

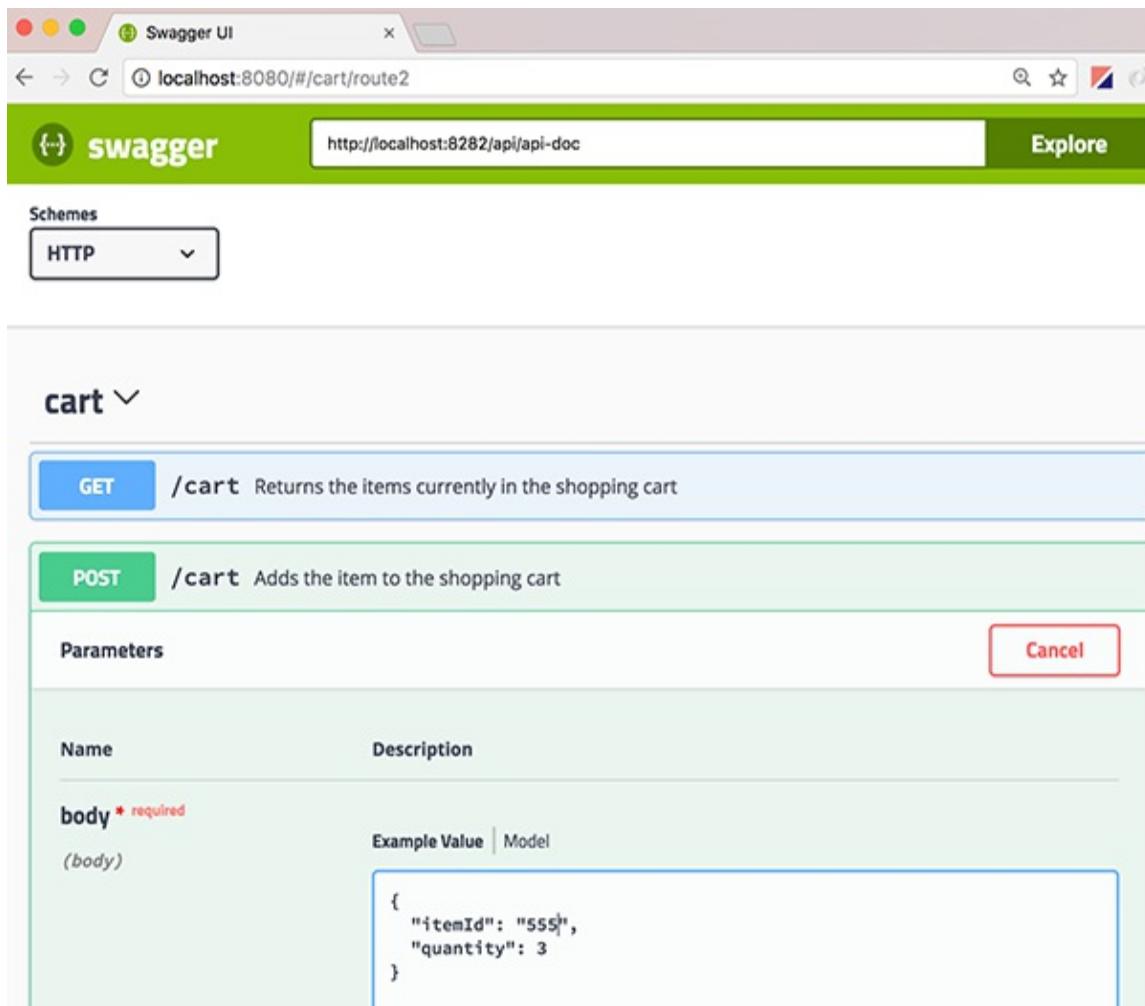


Figure 7.6 Using the Swagger UI to explore the shopping cart REST service. Here we're about to call the `POST` method to add an item to the shopping cart.

7.3.3 RULES AND INVENTORY PROTOTYPES

Not all microservices use REST services. For example, the Rider Auto Parts inventory system is accessible only by using a JMS messaging broker and XML as the data format. This means the rules microservice needs to use JMS and XML, and therefore you decide to use a Java EE micro container (WildFly Swarm) to host the rules microservice. This isn't bad news, because a goal of the prototype is to use a variety of technologies to learn what works

and what doesn't work.

The rules microservice depends on the inventory back end, so let's start there first.

THE INVENTORY BACK END

Because you develop the prototype from home, you don't want to access the Rider Auto Parts test environment and therefore build a quick simulated inventory back end using standalone Camel with an embedded ActiveMQ broker, which starts up using a main class. The embedded ActiveMQ broker listens for connections on its default port:

```
<transportConnector name="tcp" uri="tcp://0.0.0.0:61616"/>
```

The rules microservice can communicate using JMS to the embedded broker via localhost:61616.

The back end is implemented using the following mini Camel route:

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
    <route id="inventory">  
        <from uri="activemq:queue:inventory"/> ①
```

①

JMS request/reply to query inventory

```
        <to uri="bean:inventory"/> ②
```

②

Bean that simulates inventory back end

```
    </route>  
</camelContext>
```

The rules microservice sends a JMS message to the inventory queue ①, which then gets routed to the bean ② that returns the items in the inventory in XML format, which is returned back to

the rules microservice over JMS.

THE RULES MICROSERVICE

The rules microservice uses WildFly Swarm as the runtime with embedded Camel. The service exposes a REST service that the recommendation service will use. Because WildFly Swarm is a Java EE server, you can implement the REST service using JAX-RS technology.

This is done by creating a JAX-RS application class:

```
@ApplicationPath("/") ①
```

①

JAX-RS application using root (/) as context-path

```
public class RulesApplication extends Application {  
}
```

This implementation is short—we don't need any custom configuration. The `@ApplicationPath` annotation ① is used to define the starting context-path that the REST services will use. The REST service is implemented in the `RulesController` class, as shown in the following listing.

Listing 7.12 JAX-RS REST implementation

```
@ApplicationScoped ①
```

①

CDI application-scoped bean

```
@Path("/api") ②
```

②

JAX-RS REST service using /api as context-path

```
public class RulesController {
```

```
@Inject  
@Uri("direct:inventory") ③
```

③

Injects Camel's FluentProducerTemplate

```
private FluentProducerTemplate producer;  
  
@GET  
@Produces(MediaType.APPLICATION_JSON)  
@Path("/rules/{cartIds}") ④
```

④

HTTP GET service

```
public List<ItemDto> rules(@PathParam("cartIds") String  
cartIds) {  
    List<ItemDto> answer = new ArrayList<>();  
  
    ItemsDto inventory =  
producer.request(ItemsDto.class); ⑤
```

⑤

Calls Camel route to get items from inventory service

```
for (ItemDto item : inventory.getItems()) { ⑥
```

⑥

Filters duplicate items

```
boolean duplicate = cartIds != null  
                    && cartIds.contains("" +  
item.getItemNo());  
if (!duplicate) {  
    answer.add(item);  
}  
}
```

```
Collections.sort(answer, new ItemSorter()); ⑦
```

7

Sorts and ranks items to be returned

```
    return answer;
}
}
```

The controller class can be automatically discovered via CDI by annotating the class with `@ApplicationScoped` ①. If this isn't done, the class would need to be registered manually in the JAX-RS application class. The `@Path` annotation ② is used to denote the class as hosting REST services that are serviced from the given `/api` context path. To call the inventory back end, you'll use Camel and therefore inject `FluentProducerTemplate` ③ to make it easy to call a Camel route from within the class.

The rules method ④ is annotated with `GET` to set this up as an HTTP `GET` service returning data in JSON format. Notice that `@Path` and `@PathParam` map to the `partIds` method parameter from the HTTP URL using the `{cartIds}` syntax. For example, if the REST service is called using `HTTP GET /rules/123,456` and then `partIds` is mapped to the String value "123, 456".

The injected `FluentProducerTemplate` ③ is used to let Camel call the back end service in one line of code ⑤. Because the service is returning a response, you use the `request` method (`request = InOut, send = InOnly`) and automatically convert the response to the `ItemsDto` POJO type.

The rules microservice then filters out duplicate items that aren't intended to be returned ⑥. Finally, the items are sorted by using the `ItemSorter` implementation ⑦ that ranks the items to custom rules.

Camel is used to call the inventory service via JMS messaging, as shown in the Camel route:

```
from("direct:inventory")
    .to("jms:queue:inventory") ①
```

1

Calls the inventory back end using JMS

.unmarshal().jaxb("camelinaction"); ②

②

Converts response from XML to POJO classes using JAXB

The route uses the Camel JMS component ①, which needs to be configured. This is easily done in Java code using CDI @Produces:

```
@Produces  
@Named("jms")  
public static JmsComponent jmsComponent() {  
    ActiveMQComponent jms = new ActiveMQComponent();  
    jms.setBrokerURL("tcp://localhost:61616");  
    return jms;  
}
```

The last thing you need to implement is the mapping between XML and JSON, and you can do that using POJO classes and JAXB. The Camel route converts the returned XML data from the back end to POJO by using JAXB unmarshal ②.

The REST service in [listing 7.12](#) is declared to produce JSON, and the return type of the method is `List<ItemDto>`. That's all you need to do because WildFly Swarm will automatically convert the POJO classes to JSON using JAXB.

TIP When using JAXB, always remember to list the POJO classes in the `jaxb.index` file, which resides in the `src/main/resources` folder.

RUNNING THE RULES AND INVENTORY SERVICES

You can run these services from the `chapter7/prototype` directory. First you need to start the inventory service:

```
cd inventory  
mvn compile exec:java
```

Then from another shell, you can start the rules microservice:

```
cd rules  
mvn wildfly-swarm:run
```

From a web browser, you can call the rules service using the following URL:

```
http://localhost:8181/api/rules/123, 456
```

The inventory service has been hardcoded to return three items using IDs 123, 456, and 789. Therefore, you can try URLs such as these:

```
http://localhost:8181/api/rules/123  
http://localhost:8181/api/rules/456, 789
```

The response returned is intended to filter out items that are already present in the shopping cart, so you may have responses that don't return one or more of the item IDs provided as inputs. Only one microservice is left in the prototype to develop: the rating service.

7.3.4 RATING PROTOTYPE

It's been a long, productive weekend, and you've already built four prototypes. Only the rating service is left. Building the prototype from home, you don't have access to the provider of the rating SaaS service. Instead, you decide to build a quick-and-dirty simulation of the rating service that you can run locally. When it comes to the real deal, the rating service is hosted on Amazon AWS, and Camel has the camel-aws component that offers integration with many of the Amazon technologies.

You've used Rest DSL a few times already in this chapter, so you can quickly slash out the following code to set up a Camel REST service:

```
restConfiguration().bindingMode(RestBindingMode.json);  
①
```

1

Turns on JSON binding

```
rest("/ratings/{ids}").produces("application/json")
```

②

② GETs /ratings/{ids} REST service

```
.get().to("bean:ratingService");
```

The REST service uses JSON, so you turn on automatic JSON binding ①, which allows Camel to bind between JSON data format and POJO classes. Only one REST service rates items ②. In the interest of time, you implement the rating service to return random values:

```
@Component("ratingService")
```

①

① Naming the bean ratingService

```
public class RatingService {  
    public List<RatingDto> ratings(@Header("ids") String  
    items) {
```

②

② Parameter mapping of rating IDs

```
        List<RatingDto> answer = new ArrayList<>();  
        for (String id : items.split(",")) {  
            RatingDto dto = new RatingDto();  
            answer.add(dto);  
            dto.setItemNo(Integer.valueOf(id));  
            dto.setRating(new Random().nextInt(100));
```

③

Generating random rating value

```
}
```

```
return answer;
```

```
    }  
}
```

The class is annotated with `@Component("ratingService")` ❶, which allows Spring to discover the bean at runtime and allows Camel to call the bean by its name, which is done from the Camel route using `to("bean:ratingService")`. The rating method ❷ maps to the Rest DSL `/ratings/{ids}` path using the `@Header("ids")` annotation, which ensures that Camel will provide the values of the IDs upon calling the bean.

The returned value from the bean is `List<RatingDto>`, which is a POJO class. The POJO class is merely a class with two fields:

```
public class RatingDto {  
    private int itemNo;  
    private int rating;  
    // getter and setter omitted  
}
```

Because you've turned on automatic JSON binding in the Rest DSL configuration, Camel will automatically convert from `List<RatingDto>` to a JSON representation.

RUNNING THE RATING SERVICE

You can find the rating prototype in the `chapter7/prototype/rating` directory, which you can run using the following Maven goal:

```
mvn spring-boot:run
```

Then from a web browser you can open the following URL:

```
curl http://localhost:8383/api/ratings/123,456
```

Phew! You've now implemented prototypes for all the microservices for the Rider Auto Parts recommendation system. So far, you've been running each service in isolation. It's about time that you see how they work together.

7.3.5 PUTTING ALL THE MICROSERVICES

TOGETHER

During a weekend, you've been able to build a prototype for the Rider Auto Parts recommendation system that consists of five independent microservices, as previously illustrated in [figure 7.3](#). Now it's time to put all the pieces together, plug in the power, and see what happens.

The book's source code contains the entire prototype in the `chapter7/prototype` directory. You need to open five command shells at once, and for each shell run the following commands in the given order:

1. Start the shopping cart service:

```
cd cart  
mvn compile exec:java
```

2. Start the inventory service:

```
cd inventory  
mvn compile exec:java
```

3. Start the rules service:

```
cd rating  
mvn spring-boot:run
```

4. Start the recommendation service:

```
cd rules  
mvn wildfly-swarm:run
```

5. Start the rating service:

```
cd recommend  
mvn spring-boot:run
```

Then you can open a web browser and call the recommendation service using the following URL:

```
http://localhost:8080/api/recommend
```

The response should contain three items being recommended. That may seem like no big deal, but you have five Java

applications running isolated in their own JVM, integrated together to implement a business solution.

The weekend is coming to a close, and you want to summarize what you've learned so far.

WHAT YOU'VE LEARNED

First, you've learned that it's good from time to time to clear your schedule and set aside one or two days to let you fully concentrate when learning new skill sets. The busy workday and the rapid evolution of software are having tremendous impacts on business. Many companies are getting more and more worried about becoming "uberized" (beaten by new, mobile-based businesses). As a forward thinker, you take the matter into your own hands to ensure that you up your game and don't become obsolete in the job market.

With that in mind, you also gained a fair amount of hands-on experience building small microservices with various technologies. For example, you have no worries about Camel, as it fits in any kind of Java runtime of your choosing. You learned that Spring Boot is a great opinionated framework for building microservices. The curated starting dependencies are a snap to install, and Spring Boot's autoconfiguration makes configuration consistent and easy to learn and use. Camel works well with Spring Boot, and you have great power with testing as well (more on that to come in chapter 9). You dipped your toes into WildFly Swarm and learned a few things. It's a small and fast application server on which you can easily include only the Java EE parts you need. But WildFly Swarm doesn't yet have as good a story as Spring Boot when it comes to configuration management, and you struggled a bit with configuring WildFly Swarm and your Camel applications in a seamless fashion. You learned that using Camel with WildFly requires knowledge of CDI, which is a similar programming model to Spring's annotations, so the crossover is fairly easy to learn. You want to do a few more things before the family returns and your weekend comes to a

close. At the beginning of this chapter, you learned about some of the characteristics of microservices, which you want to compare with what you've done this weekend. Your comments are marked in table 7.1.

Table 7.1 Comments about microservices characteristics

Characteristic	Comment
Small in size	Each microservice is surely small in size and does one thing and one thing only.
Observabile	Microservices should be observable from centralized monitoring and management. This is covered in chapter 16.
Design for failure	An important topic that we haven't covered sufficiently in this chapter. But there's more to come in chapter 11, covering error handling.
Highly configurable	Spring Boot is highly configurable. WildFly Swarm doesn't yet offer configuration on the same level as Spring Boot. But you can use the Camel properties component for Camel-only configuration.
Smart endpoints and dumb routes	Can easily be done using Camel in smaller Camel routes.
Testable	Camel has great support for testing, which is covered in chapter 9.

You've come to realize that you haven't focused so much on the aspect of designing for failure. You still have the five microservices running, so you quickly play the devil's advocate and stop the inventory service, and then refresh the web browser. Kaboom—the recommendation service fails with an HTTP error 500, and you find errors logged in the consoles. Oh boy—how could you forget about this?

When building microservices or distributed applications, it's paramount to assume that failures happen, and you must design with that in mind. Designing for and dealing with failures is covered in chapters 11, 17, and 18. But we won't leave you in the dark—let's take a moment to lay out the landscape and show you

how to use a design pattern to handle failures, and then we'll return to this prototype before we reach the end of this chapter.

7.4 Designing for failures

Murphy's law—*Whatever can go wrong will go wrong*—is true in a distributed system such a microservice architecture. You could spend a lot of time preventing errors from happening, but even so, you can't predict every case of how microservices could fail in a distributed system. Therefore, you must face the music and design your microservices as resilient and fault-tolerant. This section touches on the following patterns you can use to deal with failures:

- Retry
- Circuit Breaker
- Bulkhead

The first pattern, Retry, is the traditional solution to dealing with failures: if something fails, try again. The second and third are patterns that have become popular with microservice architectures; they're often combined, and you get this combination with the Netflix Hystrix stack.

But let's start from the top with the traditional way.

7.4.1 USING THE RETRY PATTERN TO HANDLE FAILURES

The Retry pattern is intended for handling transient failures, such as temporary network outages, by retrying the operation with the expectation that it'll then succeed. The Camel error handler uses this pattern as its primary functionality. This section only briefly touches on the Retry pattern and Camel's error handler because chapter 11 is entirely devoted to this topic. But you can use the Retry pattern in the rules prototype containing the following Camel route that calls the inventory service:

```
public class InventoryRoute extends RouteBuilder {  
    public void configure() throws Exception {  
        from("direct:inventory")  
            .to("jms:queue:inventory") ❶  
    }  
}
```

❶

Calling inventory microservice, which may fail

```
    .unmarshal().jaxb("camelinaction");  
}  
}  
}
```

To handle failures when calling the inventory microservice ❶, we can configure Camel's error handler to retry the operation:

```
public void configure() throws Exception {  
    errorHandler(defaultErrorHandler()  
        .maximumRedeliveries(5) ❶  
    )  
}
```

❶

Configures error handler to retry up to 5 times

```
    .redeliveryDelay(2000));  
  
    from("direct:inventory")  
        .to("jms:queue:inventory")  
        .unmarshal().jaxb("camelinaction");  
}
```

As you can see, you've configured Camel's error handler to retry ❶ up to five times, with a two-second delay between each attempt. For example, if the first two attempts fail, the operation succeeds at the third attempt and can continue to route the message. Only if all attempts fail does the entire operation fail, and an exception is propagated back to the caller from the Camel route.

If only the world were so easy. But you must consider these five factors when using the Retry pattern.

WHICH FAILURES

Not all failures are candidates for retrying. For example, network operations such as HTTP calls or database operations are good candidates. But in every case, it depends. An HTTP server may be unresponsive, and retrying the operation in a bit may succeed. Likewise, a database operation may fail because of a locking error due to concurrent access to a table, which may succeed on the next retry. But if the database fails because of wrong credentials, retrying the operation will keep failing, and therefore it isn't a candidate for retries.

Camel's error handler supports configuring different strategies for different exceptions. Therefore, you can retry only network issues, and not exceptions caused by invalid credentials. Chapter 11 covers Camel's error handler.

HOW OFTEN TO RETRY

You also have to consider how frequently you may retry an operation, as well as the total duration. For example, a real-time service may be allowed to retry an operation only a few times, with short delays, before the service must respond to its client. In contrast, a batch service may be allowed many retries with longer delays.

The retry strategy should also consider other factors, such as the SLAs of the service provider. For example, if you call the service too aggressively, the provider may throttle and degrade your requests, or even blacklist the service consumer for a period of time. In a distributed system, a service provider must build in such mechanisms—otherwise, consumers of the services can overload their systems or degrade downstream services. Therefore, the service provider often provides information about the remaining request count allowed. The retry strategy can read the returned request count and attempt only within the permitted parameters.

TIP Camel’s error handler offers a `retryWhile` functionality that allows you to configure to retry only while a given predicate returns a `true` value.

IDEMPOTENCY

Another factor to consider is whether calling a service is idempotent. A service is *idempotent* if calling the service again with the same input parameters yields the same result. A classic example is a banking service: calling the bank balance service is idempotent, whereas calling the withdrawal service isn’t idempotent.

Distributed systems are inherently more complex. Because of remote networks, a request may have been processed by the remote service but not received by the client, which then assumes the operation failed, and retries the same operation, which then may cause unexpected side effects.

TIP In a distributed system, it’s difficult to ensure “once and only once” guarantees. Design with idempotency and duplicates in mind.

The service provider can use the Idempotent Consumer EIP pattern to guard its service with idempotency. We cover this pattern in chapter 12, section 12.5.

MONITORING

Monitoring and tracking retries are important. For example, if some operations have too many retries before they either fail or succeed, it indicates a problem area that needs to be fixed. Without proper monitoring in place, retries may remain unnoticed and affect the overall stability of the system.

Camel comes with extensive metrics out of the box that track

every single message being routed in fine-grained detail. You'll be able to pinpoint exactly which EIP or custom processor in your Camel applications is causing retries or taking too long to process messages. You'll learn all about monitoring your Camel applications in chapter 16.

TIP Camel uses the term *redelivery* when a message is retried.

TIME-OUT AND SLAs

When using the Retry pattern, and retries kick in, the overall processing time increases by the additional time of every retry attempt. In a microservice architecture, you may have SLAs and consumer time-outs that should be factored in. Take the maximum time allowed to handle the request as specified in SLAs and consumer times and then calculate appropriate retry and delay settings.

The example at the start of section 7.4.1 uses Camel's error handler for retries. What if you don't use Camel? How can you do retries?

7.4.2 USING THE RETRY PATTERN WITHOUT CAMEL

If you don't use Camel, you can still use the Retry pattern by using good old-fashioned Java code with a `try ... catch` loop. For example, in the recommendation service that's using Spring Boot and Java, you could implement a Retry pattern, as shown in the following listing.

[Listing 7.13](#) Retry pattern using Java `try ... catch` loop

```
String itemIds = null;
ItemDto[] items = null;
int retry = 5 + 1;
for (int i = 0; i < retry; i++) {    ①
```

1

Retry loop

```
try {
    LOG.info("Calling rules service {}", rulesUrl);
    items = restTemplate.getForObject(rulesUrl,
                                      ItemDto[].class,
                                      id, cartIds);
    itemIds = itemsToCommaString(items);
    LOG.info("Inventory items {}", itemIds);
    break;      2
```

2

Break out loop if succeeded

```
} catch (Exception e) {
    if (i == retry - 1) {
        throw e;      3
```

3

Rethrow exception if end of loop

```
}
```

In the Java code, you can implement a Retry pattern by using a for loop 1. You keep looping until either the operation succeeds 2 or keeps on failing until the end of the loop 3. As you can see, implementing this code is cumbersome, error prone, and *ugly*. Isn't there a better way? Well, if you're using Apache Camel, we've already shown you the elegancy of Camel's error handler; but some of our microservices aren't using Camel, so what you can do? You can use a pattern called Circuit Breaker, which has become popular with microservice architectures.

7.4.3 USING CIRCUIT BREAKER TO HANDLE FAILURES

The previous section covered the Retry pattern and the ways it

can overcome transient errors, such as slow networks, or a temporary overloaded service, by retrying the same operation. But sometimes failures aren't transient—for example, a downstream service may not be available, may be overloaded, or may have a bug in the code, causing application-level exceptions. In those situations, retrying the same operation and putting additional load on an already struggling system provide no benefit. If you don't deal with these situations, you risk degrading your own service, holding up threads, locks, and resources, and contributing to cascading failures that can degrade the overall system, and even take down a distributed system.

Instead, the application should detect that the downstream service is unhealthy and deal with that in a graceful manner until the service is back and healthy again. What you need is a way to detect failures and fail fast in order to avoid calling the remote service until it's recovered.

The solution for this problem is the Circuit Breaker pattern, described by Michael Nygard in *Release It!* (Pragmatic Bookshelf, 2007).

CIRCUIT BREAKER PATTERN

The Circuit Breaker pattern is inspired by the real-world electrical circuit breaker, which is used to detect excessive current draw and fail fast to protect electrical equipment. The software-based circuit breaker works on the same notion, by encapsulating the operation and monitoring it for failures. The Circuit Breaker pattern operates in three states, as illustrated in figure 7.7.

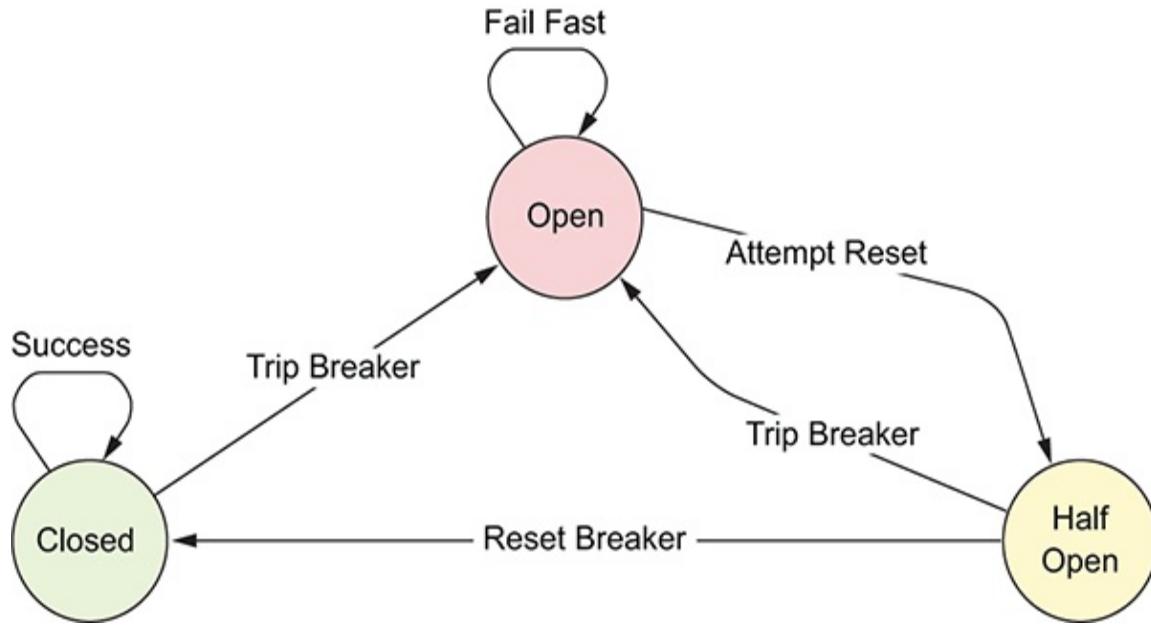


Figure 7.7 The Circuit Breaker pattern operates in three states. In the closed state (green), the operation is successful. Upon a number of successive failures, the breaker trips into the open state (red) and fails fast. After a short period, the breaker attempts to reset in the half-open state (yellow). If it's successful, the break resets into a closed state (green), and in case of failure transfers back to an open state (red).

The states are as follows:

- *Closed*—When operating successfully.
- *Open*—When failure is detected and the breaker opens to short-circuit and fail fast. In this state, the circuit breaker avoids invoking the protected operation and avoids putting additional load on the struggling service.
- *Half Open*—After a short period in the open state, an operation is attempted to see whether it can complete successfully, and depending on the outcome, it will transfer to either open or closed state.

Speaking from practical experience with using the Circuit Breaker pattern, you may mix up the open and closed states. It takes a while to make your brain accept the fact that closed is good (green) and open is bad (red).

At first sight, the Retry and Circuit Breaker patterns may seem similar. Both make an operation resilient to failures from

downstream services. But the difference is that the Retry pattern invokes the same operation in hopes it will succeed, whereas the Circuit Breaker prevents overloading struggling downstream systems. The former is good for dealing with transient failures, and the latter for more permanent and long-lasting failures.

In microservice architecture and distributed systems, the Circuit Breaker pattern has become a popular choice. In more traditional systems, the Retry pattern has been the primary choice. But one doesn't exclude the other; both patterns can be used to deal with failures.

What support does Camel have for circuit breakers?

CIRCUIT BREAKER AND CAMEL

Camel offers two implementations:

- Load Balancer with Circuit Breaker
- Netflix Hystrix

The first implementation of the Circuit Breaker pattern is an extension to the Load Balancer pattern. This implementation comes out of the box from camel-core and provides a basic implementation. But recently the Netflix Hystrix library has become popular, so the Camel team integrated Hystrix as a native EIP pattern in Camel. We recommend using the latter, which is also the implementation covered in this book.

7.4.4 NETFLIX HYSTRIX

Hystrix is a latency and fault-tolerant library designed to isolate points of access to remote services and stop cascading failures in complex distributed systems where failures are inevitable.

Hystrix provides the following:

- Protection against services that are unavailable
- Time-out to guard against unresponsive services
- Separation of resources using the Bulkhead pattern

- Shredding loads and failing fast instead of queueing
- Self-healing
- Real-time monitoring

We'll describe each of these bullet items, but let's start with a simple Hystrix example.

USING HYSTRIX WITH PLAIN JAVA

Hystrix provides the `HystrixCommand` class, which you extend by implementing the `run` method containing your potential faulty code, such as a remote network call. The following listing shows how this is done.

Listing 7.14 Creating a custom Hystrix command

```
public class MyCommand extends HystrixCommand<String>
{ ①
```

①

Extending `HystrixCommand`

```
    public MyCommand() {
        super(HystrixCommandGroupKey.Factory.asKey("MyGroup")); ②
```

②

Specifying group

}

```
    protected String run() throws Exception { ③
```

③

All unsafe code goes into run method

```
COUNTER++;
if (COUNTER % 5 == 0) {
    throw new IOException("Forced error"); 4
```

4

Simulated error every fifth call

```
}
```

```
    return "Count " + COUNTER;
}
}
```

As you can see, we've created our custom Hystrix command in the `MyCommand` class by extending `HystrixCommand<String>` **1** with `String` specified as the type, which then corresponds to the return type of the `run` method **3**. Every Hystrix command must be associated with a group, which we configured in the constructor **2**. All the potentially unsafe and faulty code is put inside the `run` method **3**. In this example, we've built into the code a failure that happens for every fifth call to simulate some kind of error **4**.

CALLING A HYSTRIX COMMAND FROM JAVA

How do you use this command? That's simple with Hystrix: you create an instance of it and call the—no, not the `run` method—but the `execute` method, as shown here:

```
MyCommand myCommand = new MyCommand();
String out = myCommand.execute();
```

Here's the result of running this command:

```
Count 1
```

If you run it again like this

```
MyCommand myCommand = new MyCommand();
String out = myCommand.execute();
String out2 = myCommand.execute();
```

then you'd expect the following:

```
Count 1  
Count 2
```

But that isn't what happens. Instead, you have this:

```
Count 1  
com.netflix.hystrix.exception.HystrixRuntimeException:  
MyCommand command executed multiple times - this is not  
permitted
```

That's an important aspect with Hystrix: each command isn't reusable; you can use a command once and only once. You can't store any state in a command, because it can be called only once, and no state is left over for successive calls.

You can find this example in the source code, in the chapter7/hystrix directory. Try the example by using the following Maven goal:

```
mvn test -Dtest=MyCommandTest
```

You may have noticed in [listing 7.14](#) the COUNTER instance, which is used as a global state stored outside the Hystrix command, and the fact that the example is constructed to fail every fifth call. How do you deal with exceptions thrown from the run method?

ADDING Fallback

When the run method can't be executed successfully, you can use the Hystrix built-in fallback method to return an alternative response. For example, to add a fallback to [listing 7.14](#), you override the getFallback method as follows:

```
protected String getFallback() {  
    return "No Counter";  
}
```

Running this command 10 times yields the following output:

```
new MyCommand().execute() -> Count 1  
new MyCommand().execute() -> Count 2  
new MyCommand().execute() -> Count 3
```

```
new MyCommand().execute() -> Count 4
new MyCommand().execute() -> No Counter
new MyCommand().execute() -> Count 6
new MyCommand().execute() -> Count 7
new MyCommand().execute() -> Count 8
new MyCommand().execute() -> Count 9
new MyCommand().execute() -> No Counter
```

The example is constructed to fail every fifth time, which is when the fallback kicks in and returns the alternative response.

You can try this example, which is in the source code in the chapter7/hystrix directory, by using the following Maven goal:

```
mvn test -Dtest=MyCommandWithFallbackTest
```

Okay, what about using Hystrix with Camel?

7.4.5 USING HYSTRIX WITH CAMEL

Camel makes it easy to use Hystrix. All you have to do is to add the camel-hystrix dependency to your project and then use the Hystrix EIP in your Camel routes.

Let's migrate the previous example from section 7.4.4 to use Camel. The code from [listing 7.14](#), which was implemented in the run method of `HystrixCommand`, is refactored into a plain Java bean:

```
public class CounterService {
    private int counter; ①
```

①

Storing counter state in the bean

```
public String count() throws IOException { ②
```

②

The unsafe code

```
counter++;
if (counter % 5 == 0) {
```

```

        throw new IOException("Forced error");
    }
    return "Count " + counter;
}
}

```

The code that was previously in the `run` method of `HystrixCommand` is migrated as is in the `count` method ②. We can now store the counter state directly in the bean ① because the bean isn't a `HystrixCommand` implementation. From section 7.4.4, you learned that it's a requirement for a Hystrix command to be stateless and executable only once, which forced us to store any state outside the `HystrixCommand`.

Using Hystrix in Camel routes is so easy, you need just a few lines of code:

```

public void configure() throws Exception {
    from("direct:start")
        .hystrix() ①

```

①

Hystrix EIP

```

        .to("bean:counter")
    .end() ②

```

②

End of Hystrix block

```

    .log("After calling counter service: ${body}");
}

```

In the route, you use Hystrix to denote the beginning of the protection of Hystrix. Any of the following nodes are run from within `HystrixCommand`. In this example, that would be calling the counter bean. To denote when the Hystrix command ends, you use `end` ② in Java DSL, which means the log node is run outside Hystrix.

You can have as many service calls and EIPs as you like inside

the Hystrix block. For example, to call a second bean, you can do this:

```
.hystrix()
    .to("bean:counter")
    .to("bean:anotherBean")
.end()
```

Using Hystrix in XML is just as easy. The equivalent route is shown in the following listing.

Listing 7.15 Using Hystrix EIP using XML DSL

```
<bean id="counterService"
class="camelaction.CounterService"/> ①
```

①

Defines counter bean

```
<camelContext
xmlns="http://camel.apache.org/schema/spring">
<route>
    <from uri="direct:start"/>
    <hystrix> ②
```

②

Hystrix EIP

```
        <to uri="bean:counterService"/>
    </hystrix> ③
```

③

End of Hystrix block

```
        <log message="After calling counter service:
${body}"/>
    </route>
</camelContext>
```

In XML DSL, we first define the counter service bean as a Spring

<bean> ❶ In the Camel route, you can easily use Hystrix by using <hystrix> ❷. The nodes inside the Hystrix block ❸ are then run under the control from a `HystrixCommand`.

What about using fallbacks?

ADDING Fallback

Using a Hystrix fallback with Camel is also easy:

```
public void configure() throws Exception {  
    from("direct:start")  
        .hystrix()  
            .to("bean:counter")  
        .onFallback() ❶
```

❶

Hystrix fallback

```
    .transform(constant("No Counter"))  
    .end()  
    .log("After calling counter service: ${body}");  
}
```

The fallback is added by using `.onFallback()` ❶, and the following nodes are included. In this example, we perform a message transformation to set the message body to the constant value of `No Counter`. Using fallback with XML DSL is easy as well:

```
<route>  
    <from uri="direct:start"/>  
    <hystrix>  
        <to uri="bean:counterService"/>  
        <onFallback>
```

Hystrix fallback

```
    <transform>  
        <constant>No Counter</constant>  
    </transform>  
</onFallback>
```

```
</hystrix>
<log message="After calling counter service: ${body}" />
</route>
```

You can find this example with the source code in the chapter7/hystrix-camel directory. Try it using the following Maven goals:

```
mvn test -Dtest=CamelHystrixWithFallbackTest
mvn test -Dtest=SpringCamelHystrixWithFallbackTest
```

One aspect we haven't delved into is that each `HystrixCommand` must be uniquely defined using a command key. Hystrix enforces this in the constructor of `HystrixCommand`, as in [listing 7.14](#) when we weren't using Camel.

CONFIGURING COMMAND KEY

Each Hystrix EIP in Camel is uniquely identified via its node ID, which by default is the Hystrix command key. For example, the following route

```
.hystrix()
    .to("bean:counter")
    .to("bean:anotherBean")
.end()
```

would use the autoassigned node IDs, which typically are named using the node and a counter such as `hystrix1`, `bean1`, `bean2`. The command key of the Hystrix EIP would be `hystrix1`. If you want to assign a specific key, you need to configure the node ID with the value shown in bold:

```
.hystrix().id("MyHystrixKey")
    .to("bean:counter")
    .to("bean:anotherBean")
.end()
```

In XML DSL, you'd do as follows:

```
<hystrix id="MyHystrixKey">
    <to uri="bean:counterService"/>
    <to uri="bean:anotherBean"/>
```

```
</hystrix>
```

Speaking of configuring Hystrix, it has a wealth of configuration options that you can use to tailor all kind of behaviors. Some options are more important to know than others, which you'll learn about through the Bulkhead pattern.

But first let's cover the basics of configuring Hystrix.

7.4.6 CONFIGURING HYSTRIX

Configuring Hystrix isn't a simple task. Many options are available and you can tweak all kinds of details related to the way the circuit breaker should operate. All these options play a role in allowing fine-grained configuration. Your first encounters with these options will leave you baffled, and learning the nuances between the options and the role they play takes time. As a rule of thumb, the out-of-the box settings are a good compromise, and you'll most often use only a few options. This section shows you how to configure Hystrix in pure Java and as well with Camel and Hystrix.

CONFIGURING HYSTRIX WITH JUST JAVA

You configure `HystrixCommand` in the constructor by using a *somewhat special* fluent builder style. For example, to configure a command to open the circuit breaker if you get 10 or more failed requests within a rolling time period of 5 seconds, you can configure this as follows:

```
public MyConfiguredCommand() {
    super(Setter
        .withGroupKey(HystrixCommandGroupKey.Factory.asKey("MyGroup"))
        .andCommandPropertiesDefaults(
            HystrixCommandProperties.Setter()
                .withCircuitBreakerRequestVolumeThreshold(10)

        .withMetricsRollingStatisticalWindowInMilliseconds(5000)
    ));
}
```

Yes, it takes time to learn how to configure Hystrix when using this configuration style. Plenty more options are available than shown here, and we refer you to the Hystrix documentation (<https://github.com/Netflix/Hystrix/wiki>) to learn more about every option you can use.

Configuring Hystrix in Camel is easier.

CONFIGURING HYSTRIX WITH CAMEL

The previous example can be configured in Camel as follows:

```
.hystrix()  
    .hystrixConfiguration() ①
```

①

Hystrix configuration

```
    .circuitBreakerRequestVolumeThreshold(10)  
    .metricsRollingPercentileWindowInMilliseconds(5000)  
.end() ②
```

②

End of Hystrix configuration

```
    .to("bean:counter")  
.end()
```

End of Hystrix EIP

As you can see, Hystrix is configured using `hystrixConfiguration()` ①, which provides type-safe DSL with all the possible options. Pay attention to how `.end()` ② is used to mark the end of the configuration.

In XML DSL, you configure Hystrix using the `<hystrixConfiguration>` element:

```
<hystrix id="MyGroup">  
    <hystrixConfiguration
```

```

        circuitBreakerRequestVolumeThreshold="10"

metricsRollingStatisticalWindowInMilliseconds="5000"/>
<to uri="bean:counterService"/>
</hystrix>
```

As mentioned, Hystrix has a lot of options, but let's take a moment to highlight the options we think are of importance.

IMPORTANT HYSTRIX OPTIONS

Table 7.2 lists the Hystrix options we think are worthwhile to learn to use first.

Table 7.2 Important Hystrix options

Option	Default value	Description
commandKey		Identifies the Hystrix command. This option can't be configured but is locked down to be the node ID to make the command unique.
groupKey	cameleonystrix	Identifies the Hystrix group being used by the EIP to correlate statistics, circuit breaker, properties, and so forth.
circuitBreakerRequestVolumeThreshold	20	Sets the minimum number of requests in a rolling window that will trip the circuit. If below this number, the circuit won't trip, regardless of error percentage.
circuitBreakerErrorThresholdPercentage	50	Indicates the error percentage threshold (as a whole number, such as 50), at which point the circuit breaker will trip open and reject requests. It will stay tripped for the duration defined in the

percentage		circuitBreakerSleepWindowInMilliseconds option.
circuitBreakerSleepWindowInMilliseconds	5 0 0 0	Sets the time in milliseconds after a circuit breaker trips open that it should wait before trying requests again.
metricsRollingStatisticalWindowInMilliseconds	1 0 0 0 0	Indicates the duration of the statistical rolling window in milliseconds. This is how long metrics are kept for the thread pool.
corePoolSize	1 0	Sets the core thread-pool size. This is the maximum number of Hystrix commands that can execute concurrently.
maxQueueSize	- 1	Sets the maximum queue size of the thread-pool task queue.
executionTimeoutInMilliseconds	1 0 0 0	Indicates the time in milliseconds at which point the command will time out and halt execution.

Some of these options are covered in the next section about the Bulkhead pattern.

7.4.7 BULKHEAD PATTERN

Imagine that a downstream service is under pressure, and your application keeps calling the downstream service. The service is becoming latent, but not enough to trigger a time-out in your application nor trip the circuit breaker. In such a situation, the latency can stall (or appear to stall) all worker threads and cascade the latency all the way back to users. You want to limit the latency to only the dependency that's causing the slowness without consuming all the available resources in your application, process, or compute node.

The solution to this is the Bulkhead pattern. A *bulkhead* is a separation of resources such that one set of resources doesn't impact others. This pattern is well known on ships as a way to create watertight compartments that can contain water in the case of a hull breach. A famous tragedy is the *Titanic*, which had bulkheads that were too low and didn't prevent water from

leaking into adjacent compartments, eventually causing the ship to sink.

Hystrix implements the Bulkhead pattern with thread pools. Each Hystrix command is allocated a thread pool. If a downstream service becomes latent, the thread pool can become fully utilized and fail fast by rejecting new requests to the command. The thread pools are isolated from each other, so other commands can still operate successfully.

By default, the bulkhead is enabled, and each command is assigned a thread pool of 10 worker threads. The thread pool has no task pool as backlog, so if all 10 threads are used, Hystrix will reject processing new requests and fail fast.

The option `corePoolSize` can be used to configure the number of threads in the pool, and the task queue is enabled by setting the `maxQueueSize` option to a positive size.

If the additional thread pools are a concern, Hystrix can implement the bulkhead on the calling thread with counting semaphores instead. Refer to the Hystrix documentation for more information.

Another powerful feature from Hystrix is the execution time-out.

TIME-OUT WITH HYSTRIX

Every Hystrix command is executed within the scrutiny of a time-out period. If the command doesn't complete within the given time, Hystrix will react and cause the command to fail. This time-out is 100% controlled by Hystrix and shouldn't be mistaken for any time-outs from Camel components or downstream services. Hystrix comes with a time-out out of the box. The default value is sensitive at only 1 second, which is a low time-out. Therefore, you may find yourself often configuring this number to a value that suits your needs.

The time-out is listed last in table [7.2](#) and can be configured as shown in Java DSL:

```
from("direct:start")
    .hystrix()
    .hystrixConfiguration().executionTimeoutInMilliseconds(2000)
).end()
```

2 seconds as time-out

```
    .toD("direct:${body}")
).end();
```

Here's the configuration in XML DSL:

```
<route>
    <from uri="direct:start"/>
    <hystrix>
        <hystrixConfiguration
executionTimeoutInMilliseconds="2000"/>
```

2 seconds as time-out

```
        <toD uri="direct:${body}"/>
    </hystrix>
</route>
```

The book's source code includes an example of using Hystrix to call downstream Camel routes that process either 1 or 3 seconds before responding. This demonstrates that Hystrix with a time-out value of 2 seconds will succeed when using a fast response, and will fail with a time-out exception when using a slow response. You can try the example, located in the chapter7/hystrix-camel directory, using the following Maven goals:

```
mvn test -Dtest=CamelHystrixTimeoutTest
mvn test -Dtest=SpringCamelHystrixTimeoutTest
```

When a Hystrix time-out occurs, the Hystrix command is regarded as failed, and the Camel route fails with a time-out exception.

TIME-OUT AND FALLBACK WITH HYSTRIX

If a time-out error happens, you may want to let Hystrix handle this by a fallback to set up an alternative response. This can easily be done with Camel by using `onFallback`:

```
from("direct:start")
    .hystrix()
        .hystrixConfiguration()
            .executionTimeoutInMilliseconds(2000)
        .end()
        .log("Hystrix processing start: ${threadName}")
        .toD("direct:${body}")
        .log("Hystrix processing end: ${threadName}")
    .onFallback()
```

Hystrix fallback when any error happens

```
.log("Hystrix fallback start: ${threadName}")
.transform().constant("Fallback response")
.log("Hystrix fallback end: ${threadName}")
.end()
.log("After Hystrix ${body}");
```

You can find this example with the source code in the `chapter7/hystrix-camel` directory, which can be run with the following Maven goals:

```
mvn test -Dtest=CamelHystrixTimeoutAndFallbackTest
mvn test -Dtest=SpringCamelHystrixTimeoutAndFallbackTest
```

If you run the example, the fast and slow response tests will output the following to the console.

The fast response logs:

```
2017-10-01 20:21:14,654 [-CamelHystrix-1] INFO route1
- Hystrix processing start: hystrix-CamelHystrix-1
2017-10-01 20:21:14,660 [-CamelHystrix-1] INFO route2
- Fast processing start: hystrix-CamelHystrix-1
2017-10-01 20:21:15,662 [-CamelHystrix-1] INFO route2
- Fast processing end: hystrix-CamelHystrix-1
2017-10-01 20:21:15,663 [-CamelHystrix-1] INFO route1
- Hystrix processing end: hystrix-CamelHystrix-1
2017-10-01 20:21:15,666 [main] INFO route1
```

- After Hystrix Fast response

And the slow response logs:

```
2017-10-01 20:23:08,035 [-CamelHystrix-1] INFO  route3
- Slow processing start: hystrix-CamelHystrix-1
2017-10-01 20:23:10,023 [HystrixTimer-1 ] INFO  route1
- Hystrix fallback start: HystrixTimer-1
2017-10-01 20:23:10,023 [HystrixTimer-1 ] INFO  route1
- Hystrix fallback end: HystrixTimer-1
2017-10-01 20:23:10,026 [main           ] INFO  route1
    - After Hystrix Fallback response
```

Why do we show you these log lines? The clue is the information highlighted in bold. That information shows the name of the thread that's processing. In the test with the fast response, it's the same thread, `camelHystrix-1`, that routes the Camel message in the Hystrix command (Hystrix EIP). The last log line is outside the Hystrix EIP and therefore outside the Hystrix command, so it's the caller thread that's processing the message again, which in the example is the `main` thread that started the test.

On the other hand, in the slow test, the log shows that Hystrix is using two threads to process the message, `CamelHystrix-1` and `HystrixTimer-1`. This is the Bulkhead pattern in action, where the run and fallback methods from the Hystrix command are separated and being processed by separated threads.

Figure 7.8 illustrates this principle.

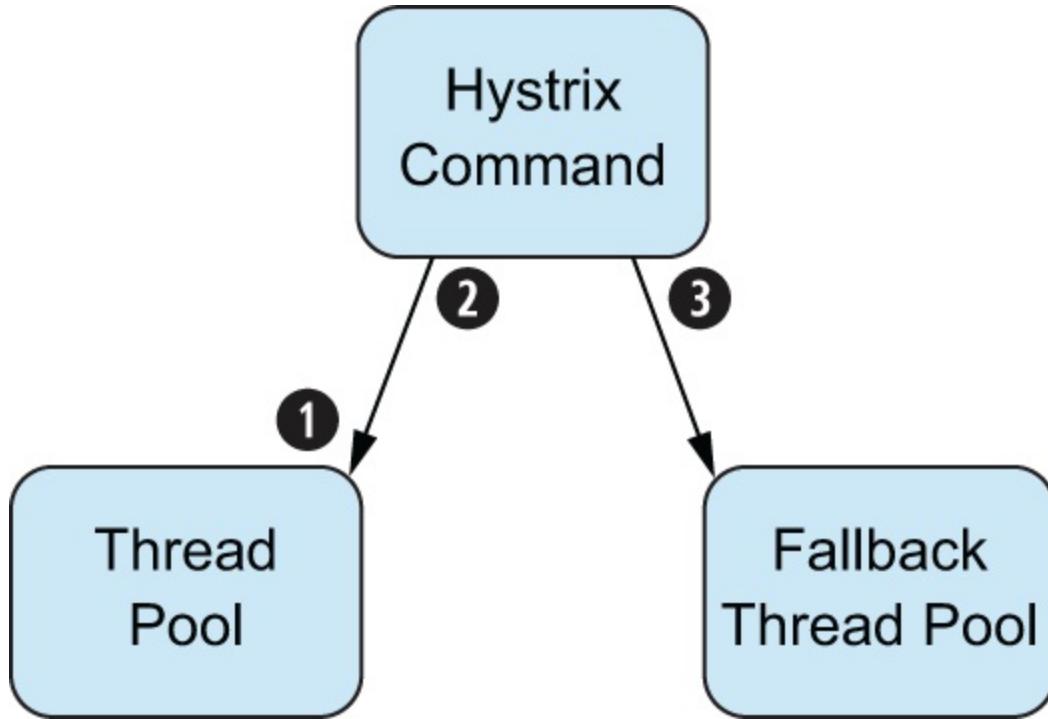


Figure 7.8 The Hystrix command uses a thread from the regular thread pool ① to route the Camel message. The task runs, and upon completion, the command completes successfully ②. But the task can also fail because of a time-out or an exception thrown during Camel routing, which causes the Hystrix command to run the fallback using a thread from the fallback thread pool ③. This ensures isolation between run and fallback, adhering to the bulkhead principle.

Hystrix fallbacks have one important caveat. In the example, you may have wondered why the fallback thread name is named `HystrixTimer-1`. That's because Hystrix triggered a time-out and executed the fallback. This time-out thread is from the time-out functionality within Hystrix that's shared across all the Hystrix commands.

Now we're down to the caveat: your Hystrix fallbacks should be written as short methods that do simple in-memory computations without any network dependency.

TIP Hystrix offers live metrics of all the states of the commands and their thread pools. This information can be visualized using the Hystrix dashboard, which is covered in section 7.4.10.

FALLBACK VIA NETWORK

If the fallback must do a network call, you have to use another Hystrix command that comes with its own thread pool to ensure total thread isolation. With Camel, this is easy, because we've implemented a special fallback for network calls that's named `onFallbackViaNetwork`:

```
from("jetty:http://0.0.0.0/myservice")
    .hystrix()
        .to("http://server-one")
    .onFallbackViaNetwork()
```

Fallback with a network call

```
    .to("http://server-backup")
```

This route exposes an HTTP service using Jetty, which proxies to another HTTP server on `http://server-one`. If the downstream service isn't available, Hystrix will fall back to another downstream service over the network at `http://server-backup`.

Fallback isn't a one-size-fits-all solution

A Hystrix fallback isn't a silver bullet that solves every problem in a microservices or distributed architecture. Whether a fallback is applicable is domain specific and determined case by case. In the Rider Auto Parts example with the recommendation service, a fallback can be useful. In this case, the recommendation service is used for displaying a personalized list for the user, but what if it isn't available or is too slow to respond? You can degrade to a generic list for users in a particular region, or just a generic list. But in other domains such as a flight-booking system, a service that performs the booking at the airline could likely not fallback because a flight seat can't be guaranteed to

have been booked with the airline.

With fresh and new knowledge in your toolbelt, you arrange for another weekend where the spouse and kids are away on a retreat. To have a good time, you've stocked up with a fresh six-pack of beer and a bottle of good scotch, just in case you run out of beer.

7.4.8 CALLING OTHER MICROSERVICES WITH FAULT-TOLERANCE

The previous prototype you built was a set of microservices that collectively implemented a recommendation system for Rider Auto Parts. But the prototype wasn't designed for failures. How can you apply fault-tolerance to those microservices? This section details what you did during the weekend to add fault-tolerance to your prototype.

First, [figure 7.9](#) illustrates the collaborations among the microservices and the places where they're in trouble.

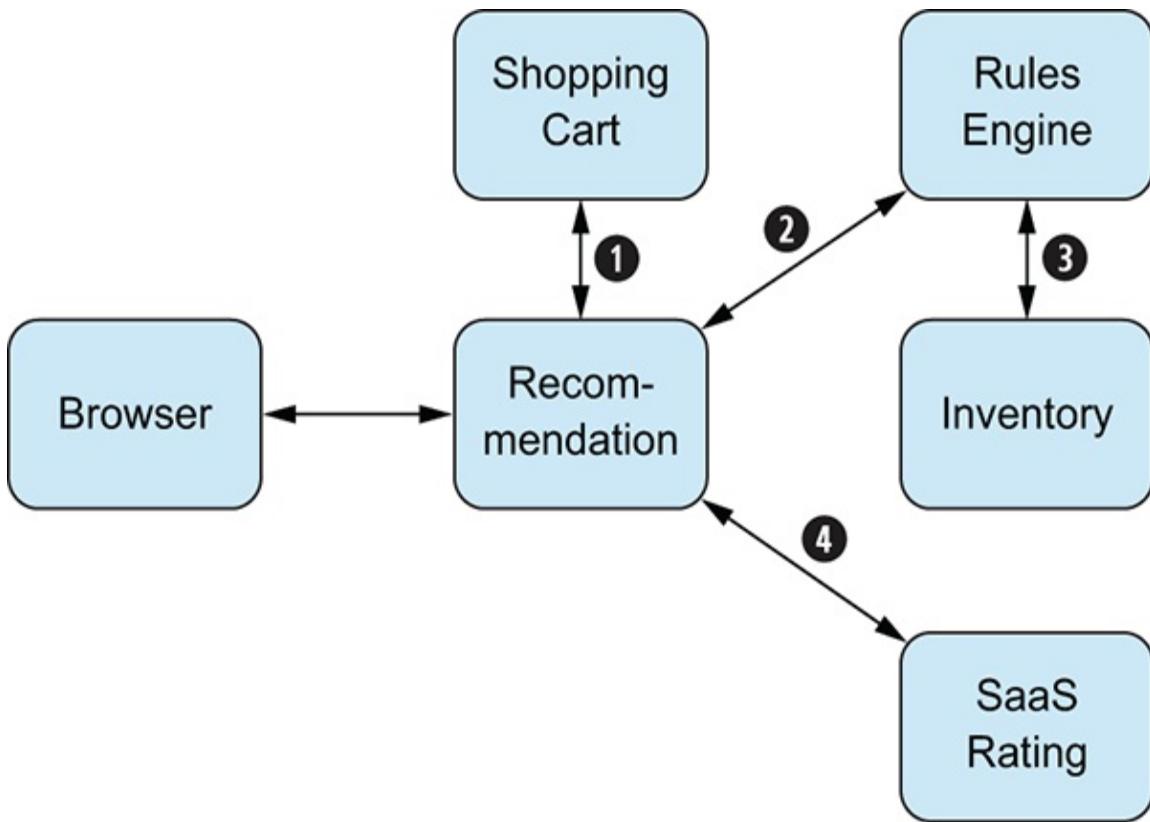


Figure 7.9 Rider Auto Parts recommendation system. Each arrow represents a dependency among the microservices. Each number represents places where you need to add fault-tolerance.

As you can see in figure 7.9, you need to add fault-tolerance four times ① ② ③ ④ to the prototype. Let's start from the top with the recommendation microservice.

USING HYSTRIX WITH SPRING BOOT

The recommendation microservice is a Spring Boot application that doesn't use Camel. To use Hystrix with Spring Boot, you need to add the following dependency to the Maven pom.xml file:

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
    <version>1.3.5.RELEASE</version>
</dependency>

```

The support for Hystrix comes with the Spring Cloud project,

hence the name of the dependency.

Support for Hystrix on Spring Boot requires you to implement an @Service class, which holds the code to run as a Hystrix command. The following listing shows what you did to add fault-tolerance at number ❶ in figure 7.9.

Listing 7.16 Using Hystrix in Spring Boot as an @Service class

@Service

❶

❶

Annotates class as a Spring @service class

```
public class ShoppingCartService {  
    private final RestTemplate restTemplate = new  
    RestTemplate();  
    @HystrixCommand(fallbackMethod = "emptyCart")
```

❷

❷

Marks method as Hystrix command

```
public String shoppingCart(String cartUrl, String id) {  
    CartDto[] carts = restTemplate.getForObject(cartUrl,  
        CartDto[].class, id);
```

❸

❸

Calls downstream service

```
String cartIds = Arrays.stream(carts)  
    .map(CartDto::toString)  
    .collect(Collectors.joining(", "));
```

❹

❹

Converts response to a comma-separated String

```
    return cartIds;
}

public String emptyCart(String cartUrl, String id)
{ 5
```

5

Method used as Hystrix fallback

```
    return "";
}
```

Here you create a new class, `ShoppingCartService`, which is responsible for calling the downstream shopping cart service. The class has been annotated with `@Service` ❶ (or `@Component`), which is required by Spring when using Hystrix. The `@HystrixCommand` annotation ❷ is added on the method that will be wrapped as a Hystrix command. All the code that runs inside this method will be under the control of Hystrix. Inside this method, you call the downstream service ❸ using Spring's `RestTemplate` to perform a REST call. Having a response, you transform this into a comma-separated string using Java 8 *stream magic* ❹. The `@HystrixCommand` annotation specifies the name of the fallback method that refers to the `emptyCart` method ❺. This method does what the name says: returns an empty shopping cart as its response.

Pay attention to the method signature of the two methods in play. The fallback method must have the exact same method signature as the method with the `@HystrixCommand` annotation.

You follow the same practice to implement fault-tolerance to calling the rules and rating microservices (❷ and ❹ in [figure 7.9](#)).

The last code change needed is in the `RecommendController` class, which orchestrates the recommendation service. The following listing lists the updated source code.

[Listing 7.17](#) `RecommendController` using the fault-tolerant

services

```
@EnableCircuitBreaker
```

1

1

Enables Hystrix Circuit Breaker

```
@RestController  
@RequestMapping("/api")  
@ConfigurationProperties(prefix = "recommend")  
public class RecommendController {  
  
    private String cartUrl;  
    private String rulesUrl;  
    private String ratingsUrl;
```

```
@Autowired
```

2

2

Dependency inject services to be used

```
private ShoppingCartService shoppingCart; 2  
@Autowired 2  
private RulesService rules; 2  
@Autowired 2  
private RatingService rating; 2  
  
@RequestMapping(value = "recommend", method =  
RequestMethod.GET,  
        produces = "application/json")  
public List<ItemDto> recommend(HttpServletRequest session) {  
    String id = session.getId();  
  
    String cartIds = shoppingCart.shoppingCart(cartUrl,  
id); 3
```

3

Calls shopping cart service

```
    ItemDto[] items = rules.rules(rulesUrl, id,  
cartIds); 4
```

4

Calls rules service

```
String itemIds = itemsToCommaString(items);

RatingDto[] ratings = rating.rating(ratingsUrl,
itemIds); 5
```

5

Calls rating service

```
for (RatingDto rating : ratings) {
    appendRatingToItem(rating, items);
}
return Arrays.asList(items);
}
```

The annotation `@EnableCircuitBreaker` ❶ is used to turn on Hystrix with Spring Boot. You then have dependency-injected the three classes with the Hystrix command ❷, which will be used from the controller to call the downstream microservices. Because the logic to call those downstream services is now moved into those separate classes, the code in the recommend method is simpler. All you do is call those services in order ❸ ❹ ❺ as if they were regular Java method calls. But under the covers, those method calls will be under the wings of Hystrix, wrapped in a Hystrix command with fault-tolerance included in the batteries.

TRYING THE EXAMPLE

Now you're ready to try this in action. Run the code from the chapter7/prototype2/recommend2 directory using the following goal from Maven:

```
mvn spring-boot:run
```

Then open a web browser to <http://localhost:8080/api/recommend>, which returns the

following response:

```
[{"itemNo":999,"name":"special","description":"Special Rider Auto Part premium service","number":100,"rating":6}]
```

What you see is that when all the downstream services aren't available, the application returns a fallback response. The fallback response is created because of these conditions:

- *ShoppingCartService*—Returns an empty cart as a response
- *RulesService*—Returns the special item with number 999 as fallback
- *RatingService*—Gives each item a rating of 6 as fallback

You turn on each of these downstream services one by one and see how the response changes accordingly.

To turn on the shopping cart, you run the following Maven command from another shell:

```
cd chapter7/prototype2/cart2  
mvn compile exec:java
```

Then you call the recommendation service again from the following URL: <http://localhost:8080/api/recommend>, which outputs the following response:

```
[{"itemNo":999,"name":"special","description":"Special Rider Auto Part premium service","number":100,"rating":6}]
```

The response is similar to before, which isn't a surprise because no items have been added to the shopping cart. Then you start the rules service by executing from another shell:

```
cd chapter7/prototype2/rules2  
mvn wildfly-swarm:run
```

And by calling the recommendation service, the response is now as follows:

```
[{"itemNo":999,"name":"special","description":"Special Rider Auto Part premium service","number":100,"rating":6}]
```

Oh boy, it's still the same response. Why is that? You do what every experienced developer always does when something isn't right—you go home. But you're already home, so what can you do? You take a little break and grab a beer from the six-pack. Dear reader, you're almost at the end of this chapter, so you don't have to take a break just yet, but you're surely welcome to enjoy a refreshment while reading.

Sitting back at the desk, you do as a professional developer would do: you go look in the logs. Looking at the rules service logs, you can see the following problem:

```
2017-10-06 10:52:30,984 ERROR
[org.apache.camel.component.jms.DefaultJmsMessageListenerContainer] (Camel(camel-1) thread #1 -
TemporaryQueueReplyManager[inventory]) Could not refresh
JMS Connection for destination 'temporary' - retrying using
FixedBackOff{interval=5000, currentAttempts=62,
maxAttempts=unlimited}. Cause: Error while attempting to
add new Connection to the pool; nested exception is
javax.jms.JMSEException: Could not connect to broker URL:
tcp://localhost:61616. Reason: java.net.ConnectException:
Connection refused
```

Ah, yeah, of course, the rules service isn't fault-tolerant and fails because it can't connect to the inventory service over JMS. This is the connection at number 3 from [figure 7.9](#). Before you start adding Hystrix to the rules service, let's continue the test and start the last service, which is the rating service. From another shell, you run the following commands:

```
cd chapter7/prototype2/rating2
mvn spring-boot:run
```

By refreshing the call to the rating service, the response is now as follows:

```
[{"itemNo":999, "name":"special", "description":"Special
Rider Auto Part premium service", "number":100, "rating":24}]
```

At first glance, the response looks the same, but when you refresh the web browser to call the URL again, a subtle change occurs in the rating:

```
[{"itemNo":999,"name":"special","description":"Special  
Rider Auto Part  
premium service","number":100,"rating":98}]
```

The rating service was implemented to return a random number. Notice that the first rating was 24, and the second is 98, as highlighted in bold.

Okay, the last piece of the puzzle is to add fault-tolerance to the rules service that's implemented using WildFly Swarm with Camel. So let's get back to Camel land.

USING HYSTRIX WITH CAMEL AND WILDFLY SWARM

Your last task for the prototype is to add fault-tolerance to the rules service, which corresponds to ③ in [figure 7.9](#). This service is using Camel running on WildFly Swarm. To use Hystrix, you have to add the camel-hystrix fraction as a Maven dependency in the pom.xml file:

```
<dependency>  
    <groupId>org.wildfly.swarm</groupId>  
    <artifactId>camel-hystrix</artifactId>  
</dependency>
```

Using Hystrix with Camel is easy, as you've already learned, so all you've done is modify the Camel route as shown here:

```
public void configure() throws Exception {  
    JaxbDataFormat jaxb = new JaxbDataFormat(); ①
```

①

Sets up JAXB for XML

```
    jaxb.setContextPath("camelinaction"); ①
```

```
        from("direct:inventory")  
            .hystrix() ②
```

②

Hystrix EIP

```
.to("jms:queue:inventory")
.unmarshal(jaxb)
.onFallback() ③
```

③

Hystrix fallback

```
.transform().constant("resource:classpath:fallback-
inventory.xml");
}
```

Because the response from the inventory service is in XML format, and this route is expected to return data as a POJO, you need to set up JAXB **①**, which is later used to unmarshal from XML to POJO. The call to the inventory service is protected by the Hystrix circuit breaker **②**. In case of any failures, the fallback is executed **③**, which loads a static response from an XML file embedded in the classpath. The fallback response is the following item:

```
<items>
  <item>
    <itemNo>998</itemNo>
    <name>Rider Auto Part Generic Brakepad</name>
    <description>Basic brakepad for any
motorbike</description>
    <number>100</number>
  </item>
</items>
```

Coding this was so fast that you didn't finish the beer you opened a while back when you took the break. But after all, the coding required adding only a Maven dependency, and then three lines of code in the Camel route, and creating an XML file with the fallback response.

You've now implemented fault-tolerance using Hystrix EIP in all the microservices at the numbers **①** **②** **③** and **④** from [figure 7.9](#). Let's try the complex example.

CONTINUE TRYING THE EXAMPLE

To continue trying the prototype, you start the rules microservice again, now that it's been updated with fault-tolerance. From the shell, you type this:

```
cd chapter7/prototype2/rules2  
mvn wildfly-swarm:run
```

Then you call the recommendation service URL,
`http://localhost:8080/api/recommend`:

```
[{"itemNo":998,"name":"Rider Auto Part Generic  
Brakepad","description":"Basic brakepad for any  
motorbike","number":100,"rating":9}]
```

The response looks familiar. It's the fallback response from the rules service. But didn't we add fault-tolerance to this? Why is it doing this? Just as you grab another beer from the six-pack, you realize that the rules service calls the downstream inventory service, which is represented as number ③ in [figure 7.9](#). And it's failing with a fallback response because you haven't yet started the inventory service. You quickly open another shell and type this:

```
cd chapter7/prototype2/inventory2  
mvn compile exec:java
```

And you just happen to glance at the logs from the rules service and notice that it stopped logging stacktraces and logged that it had successfully reconnected to the message broker, which was just started as part of the inventory service:

```
2017-10-06 12:01:56,558  
INFO[org.apache.camel.component.jms.DefaultJmsMessageListenerContainer](Camel (camel-1) thread #1 -  
TemporaryQueueReplyManager[inventory]) Successfully  
refreshed JMS Connection
```

With that in mind, you're confident that by calling the recommendation service, it should produce a different response:

```
[{"itemNo":456,"name":"Suzuki  
Brakepad","description":"Basic brakepad for  
Suzuki","number":149,"rating":28},
```

```
{"itemNo":789,"name":"Suzuki Engine  
500","description":"Suzuki 500cc  
engine","number":4,"rating":80}]
```

Yay, all the microservices are up and running, and the recommendation system returns a positive response. One last thing you want to try is to see whether the rules service is fault-tolerant when the inventory service isn't available. You stop the inventory service and call the recommendation URL again, which returns the following:

```
[{"itemNo":998,"name":"Rider Auto Part Generic Brakepad",  
"description":"Basic brakepad for any  
motorbike","number":100,"rating":34}]
```

That's the fallback response from the rules service, which would be intended to return a list of known items that Rider Auto Parts always has on stock.

You still have one beer left from the six-pack and the whisky to celebrate your success. So let's finish the weekend on a high note and build the rating microservice using Spring Boot.

7.4.9 USING CAMEL HYSTRIX WITH SPRING BOOT

It's easy to use Camel and Hystrix on Spring Boot. What you want to do is to migrate the rules microservices from using WildFly Swarm with Camel and Hystrix to using Spring Boot with Camel and Hystrix.

The first thing you do is set up the Maven pom.xml file to include the Spring Boot dependencies and the following Camel starter dependencies: camel-core-starter, camel-hystrix-starter, camel-jaxb-starter, and camel-jms-starter.

The second thing is to create the REST controller, as in the following listing.

[Listing 7.18](#) Spring REST controller calling a Camel route using direct endpoint

```
@RestController
```

①

①

Spring REST controller

```
@RequestMapping("/api")  
public class RulesController {
```

```
    @Produce(uri = "direct:inventory")
```

②

②

Injects Camel ProducerTemplate

```
    private FluentProducerTemplate producer;
```

```
    @RequestMapping(value = "rules/{cartIds}",  
                    method = RequestMethod.GET, produces =  
"application/json")  
    public List<ItemDto> rules(@PathVariable String cartIds)  
{  
    ItemsDto inventory =  
producer.request(ItemsDto.class);
```

③

Calls Camel route

```
    return inventory.getItems().stream()
```

④

④

Filters, sorts, and builds response as list

```
        .filter((i) -> cartIds == null  
                || !cartIds.contains(""  
i.getItemNo()))  
        .sorted(new ItemSorter())  
        .collect(Collectors.toList());  
    }  
}
```

The REST controller uses Spring @RestController ① instead of

JAX-RS used by WildFly Swarm. The controller class uses Camel FluentProducerTemplate ❷ to call the Camel route ❸ that does the call of the downstream service using Hystrix. The response from the Camel route is then filtered, sorted, and collected as a list ❹ by using the Java 8 stream API.

The Camel route that uses the Hystrix EIP is shown in the following listing.

Listing 7.19 Camel route using Hystrix to call downstream service using JMS

@Component

❶

❶

Enables automatic component discovery

```
public class InventoryRoute extends RouteBuilder {  
  
    public void configure() throws Exception {  
        JaxbDataFormat jaxb = new JaxbDataFormat();  
        jaxb.setContextPath("camelaction");  
  
        from("direct:inventory")  
            .hystrix()  
            .to("jms:queue:inventory")  
            .unmarshal(jaxb)  
            .onFallback()  
                .transform().constant("resource:classpath:fallback-  
inventory.xml");  
    }  
}
```

Migrating the Camel route shown in [listing 7.19](#) from WildFly Swarm to Spring Boot required only one change. The class annotation uses Spring's @Component ❶ instead of WildFly Swarm using CDI's @ApplicationScoped. That's all that was needed. That's a testimony that Camel is indifferent and works on any runtime as is.

The last migration effort is to set up Spring Boot to use ActiveMQ and set up the connection to the message broker. You

do this by adding the following dependency to the Maven pom.xml file:

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

Then you set up the connection to the ActiveMQ broker in the application.properties file:

```
spring.activemq.broker-url=tcp://localhost:61616
server.port=8181
```

The other setup is to configure Spring Boot to use HTTP port 8181, which is the port number the rules service is expected to use.

With all the beers consumed, you're almost fooled by how the Camel route in [Listing 7.19](#) knows about the Spring Boot ActiveMQ broker setting we just made in the application.properties file. That's because the Camel JMS component is using Spring JMS under the hood. And Spring JMS has, not surprisingly, support for Spring Boot and can automatically wire up to the ActiveMQ broker.

You're now ready to run this example. All the other microservices have been stopped, so when you start this example, it's the only JVM running on your computer:

```
cd chapter7/prototype2/rules2-springboot
mvn spring-boot:run
```

And this time you call the rules service directly using the following URL: <http://localhost:8181/api/rules/999>, which responds with the fallback response:

```
[{"itemNo":998,"name":"Rider Auto Part Generic Brakepad",
"description":"Basic brakepad for any
motorbike","number":100,"rating":0}]
```

That's expected because the inventory service isn't running, and therefore there's no ActiveMQ broker running that the rules

service can call. Therefore, you start the inventory service by running these commands from another shell:

```
cd chapter7/prototype2/inventory2  
mvn compile exec:java
```

And you call the rules service again, which returns a different response:

```
[{"itemNo":456,"name":"Suzuki  
Brakepad","description":"Basic brakepad for  
Suzuki","number":149,"rating":0},  
 {"itemNo":123,"name":"Honda Brakepad",  
 "description":"Basic brakepad for  
 Honda","number":28,"rating":0},  
 {"itemNo":789,"name":"Suzuki Engine  
 500","description":"Suzuki 500cc  
 engine","number":4,"rating":0}]
```

You're almost done for the day but just want to test the fault-tolerance, so you stop the inventory service that hosts the ActiveMQ broker and call the service yet again, which returns the fallback response:

```
[{"itemNo":998,"name":"Rider Auto Part Generic Brakepad",  
 "description":"Basic brakepad for any  
 motorbike","number":100,"rating":0}]
```

There's still time for one more adventure before the weekend comes to a close. The prototype includes four communication points that are protected by Hystrix circuit breakers ([figure 7.9](#)). When the recommendation service is called, four service calls could potentially fail and return a fallback response. How do you know where there are problems? That's a good question, and it's related to a whole other topic about monitoring and distributed systems, which we'll cover much more in chapters 16 and 18.

After six beers, we don't want to leave you hanging, so let's embark on another little adventure before wrapping up this chapter. How can we gain insight into the state of the circuit breakers?

7.4.10 THE HYSTRIX DASHBOARD

The Hystrix dashboard is used for visualizing the states of all your circuit breakers in your applications. For this to work, you must make sure each of your applications and microservices are able to provide real-time metrics to be fed into the dashboard.

The weekend is coming to a close, and you just want to quickly try to see what it would take to set up the Hystrix dashboard to visualize one of our microservices, the rules microservice.

To be able to feed live metrics from Hystrix into the dashboard, you need to enable *Hystrix streams*. These are easily enabled when using Spring Boot by adding the following dependencies to the Maven pom.xml file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
    <version>1.3.5.RELEASE</version>
</dependency>
```

The Spring Boot Actuator enables real-time monitoring capabilities such as health checks, metrics, and support for Hystrix streams. The support for Hystrix is provided by the Spring Cloud Hystrix dependency.

The last thing you need to do is to turn on Hystrix circuit breakers in the Spring Boot application, which you can do by adding the `@EnableCircuitBreaker` annotation to your `SpringBootApplication` class:

```
@EnableCircuitBreaker
@SpringBootApplication
public class SpringbootApplication {
```

The Hystrix EIP from the Camel route is automatically integrated with Spring Cloud Hystrix. When running the microservice, the Hystrix stream is available under the `/hystrix.stream` endpoint, which maps to the following URL: <http://localhost:8181/hystrix.stream>.

You're now ready to run the microservice. First you run the inventory service that the rules microservice communicates with:

```
cd chapter7/prototype2/inventory2  
mvn compile exec:java
```

From another shell, you run the rules microservice:

```
cd chapter7/prototype2/rules2-springboot-hystrixdashboard  
mvn spring-boot:run
```

Then you test that the microservice runs as expected by calling the rules service directly from the following URL:
<http://localhost:8181/api/rules/999>.

Everything looks okay, and you're ready to have fun and install the Hystrix dashboard.

INSTALLING THE HYSTRIX DASHBOARD

The dashboard can be installed by downloading the standalone JAR from Bintray:

<https://bintray.com/kennedyoliveira/maven/standalone-hystrix-dashboard/1.5.6>.

After the download, you run the dashboard with the following command:

```
java -jar standalone-hystrix-dashboard-1.5.6-all.jar
```

Then from a web browser, open the URL:

```
http://localhost:7979/hystrix-dashboard
```

On the dashboard, type `http://localhost:8181/hystrix.stream` in the Hostname field and click the Add Stream button. After adding the stream, you click the Monitor Streams button—and behold, there are graphics.

TIP WildFly Swarm has also made it easy to install the Hystrix dashboard as a .war file on WildFly. You can find more

information at <http://wildfly-swarm.io/tutorial/hystrix/>.

Because there are no activities, the graph isn't exciting—it's all just zeros and a flat line. So let's turn up the load. To do that, you hack up a little bash script that calls the rules service repetitively 10 times per second. You run the script as follows:

```
cd chapter7/prototype2/rules2-springboot-hystrixdashboard  
chmod +x hitme.sh  
.hitme.sh
```

And now there's activity in the dashboard.

NOTE Understanding what all the numbers on the Hystrix dashboard represent takes time to learn. You can find information on the Hystrix website (<https://github.com/Netflix/Hystrix/wiki/Dashboard>) that explains the dashboard in much more detail.

After the script has been running for a while, it's all green and happy, so you go out on a limb and stop the inventory microservice to see how the dashboard reacts to failures. And after a little while, you can see the error numbers go up, as illustrated in [figure 7.10](#).

Hystrix Stream: <http://localhost:8181/hystrix.stream>

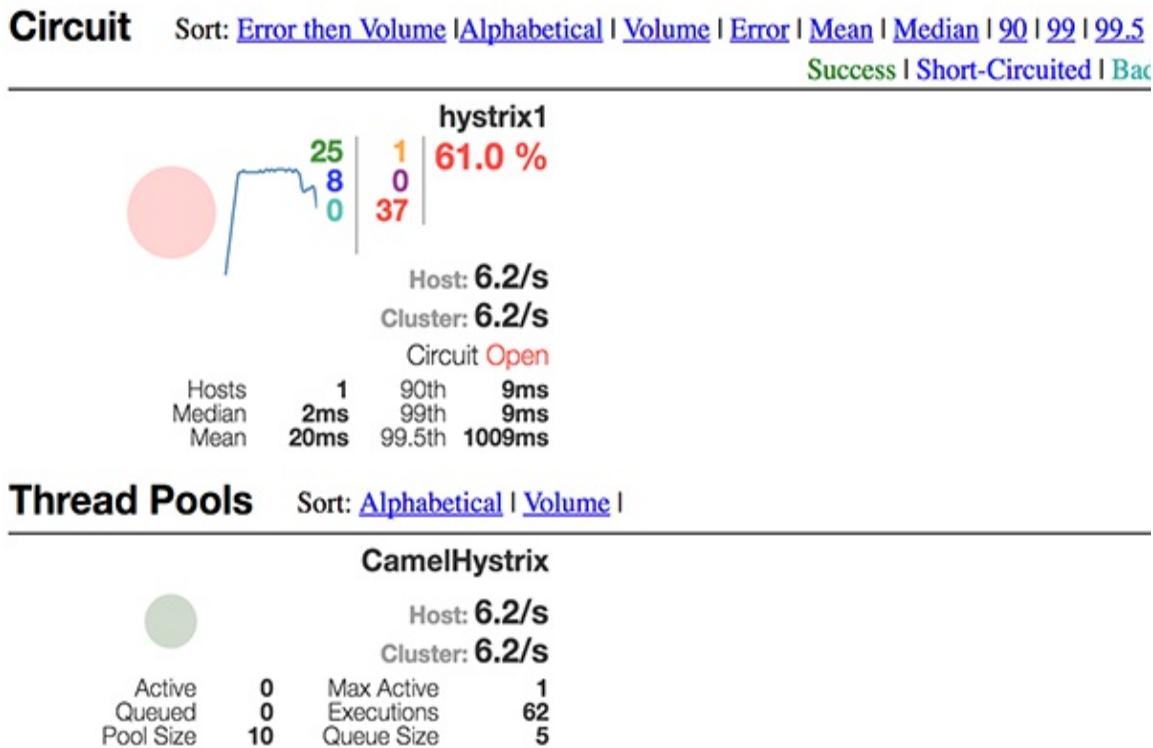


Figure 7.10 Hystrix dashboard visualizing the circuit breaker running in the rules microservice. The state of the breaker is open because of the recent number of errors as the downstream inventory service has been stopped.

Later you start the inventory service again and see the changes in the dashboard. The red numbers go down, and the green numbers up, and the state of the breaker changes from red to green as well.

The weekend is coming to an end, and tomorrow morning the family is returning. It's late in the evening, and you've done well. You pour yourself a neat glass of whisky and sit to reflect on what you've achieved this time.

WHAT YOU HAVE LEARNED THIS TIME

You've learned that in a microservices and distributed architecture, it's even more important to design for errors. With

so many individual services that communicate across the network in a mesh, errors are destined to happen. Luckily, with Camel you can easily handle errors using Camel’s error handler or Hystrix circuit breakers.

Although circuit breakers are easy to use with Camel or the `@HystrixCommand`, they’re not a silver-bullet solution for everything. The mantra about using the right tool for the right job applies. But circuit breakers are a great solution to prevent cascading errors through an entire system or apply loads on downstream systems under stress. An important setting is to apply sensitive time-outs that match your use cases. Hystrix comes with a low, one-second time setting by default, which works best for Netflix, but likely not for your organization. You certainly also learned that with circuit breakers scattered all around your individual microservices, it’s important to have good monitoring of their states. For that, you can use the Hystrix dashboard.

On the flip side, you also learned that managing and running up to six microservices on a single host/laptop is cumbersome. What you need is some kind of platform or orchestration system to manage all your running microservices. And, yeah, that’s a big and popular topic these days, which we cover in chapter 18. At first you need to master many other aspects of Camel.

The weekend is over and so is this chapter. Thanks for sticking with us all the way.

7.5 Summary and best practices

Getting to the end of this chapter was a long and bumpy ride. We hope you enjoyed the journey. Writing it was surely one of the most challenging of all the chapters, but we hope it was worth all our extra effort (and little delay in getting the book done).

Don’t get too caught up in the microservices buzzword bingo. Microservices is a great concept, but it’s not a universal super solution to every software problem. Don’t panic, because

numerous internet startups have been successful with microservices and are able to do all the jazz and deploy to production a hundred times per day. For regular enterprises, we're likely going to see a middle ground with an enterprise microservice that works the best for them.

What we've been focusing on in this book and this chapter is to cover how you can do microservice development with Camel. At the start of the chapter in section 7.1, we outlined six microservices characteristics that we believe are most applicable for Camel developers. We encourage you to take a second look at these six now that you've reached the end of this chapter. Add a bookmark and circle back to that section from time to time. We want you to have those concepts in your toolbelt.

As usual, we have put together a bulleted list with best practices and advice:

- *Microservices is more than technology*—You can't just build smaller applications and use REST as a communication protocol and think you're doing microservices. There's a lot more to master on both technical and nontechnical levels. The biggest challenges in organizations that want to move to doing microservices include potential friction from organizational structure, team communication, and old habits.
- *Small in size*—A good rule of thumb is the Jeff Bezos's two-pizza rule (where two pizzas should be enough to feed everybody). A microservice should be small enough that a single person could have its complete overview in their head.
- *Design for failures*—Microservices are inherently a distributed architecture. Every time you have remote networks, communication failures can happen. Design your microservices to be fault-tolerant. Camel offers great capabilities in this matter with its error handler and first-class support for Hystrix circuit breakers.
- *Observable*—With an increasing number of running microservices, it becomes important to be able to manage and

monitor these services in your infrastructure. Build your microservices with monitoring capabilities such as log aggregation to centralized logging. Chapter 16 covers monitoring and management.

- *Configurable*—Design your microservices to be highly configurable. This becomes an important requirement with container-based platforms, where you deploy and run your microservices as immutable containers, which means any kind of configuration of such containers must be provided externally.
- *Testable*—Microservices should be quick to test, and at best, testing should be automated as part of a continuous integration (CI)/continuous delivery (CD) pipeline. Chapter 9 covers testing.

The journey of microservices will continue in chapter 18, where we talk about microservices in the cloud.

The next chapter takes you through developing Camel projects. You may think you already have the skills to build Camel projects, and you’re surely correct. But there’s more to Camel than at first sight. Without giving away too many details, chapter 8 helps new users who are less familiar with Maven and Eclipse—to ensure that anyone can set up a development environment for building Camel projects. You’ll also learn how to build your own Camel components, data formats, and more. And last but not least, you’ll learn an important topic: how to debug your Camel routes.

8

Developing Camel projects

This chapter covers

- Managing Camel projects with Maven
- Developing Camel projects in the Eclipse IDE
- Debugging with Camel
- Creating custom components
- Using the API component framework
- Creating custom data formats

At this point, you should know a thing or two about how to develop Camel routes and how to take advantage of many Camel features. But do you know how to best start a Camel project from scratch? You could take an existing example and modify it to fit your use case, but that's not always ideal. And what if you need to integrate with a system that isn't supported out of the box by Camel?

This chapter shows you how to build your own Camel applications. We'll go over the Maven archetype tooling that allows you to skip the boring boilerplate project setup and create new Camel projects with a single command. We'll also show you how to start a Camel project from Eclipse when you need the extra power that an IDE provides. We'll even show you how to debug your new Camel application.

After that, we'll show you how to extend Camel by creating custom components. Creating custom components around large APIs isn't easy, so we'll also discuss Camel's solution to this problem: the API component framework. This framework can generate a near fully functional component just from scanning an arbitrary Java API. Finally, we'll wrap up by discussing custom data formats.

8.1 Managing projects with Maven

Camel was built using Apache Maven right from the start, so it makes sense that creating new Camel projects is easiest when using Maven. This section covers Camel's Maven *archetypes*, which are preconfigured templates for creating various types of Camel projects. After that, you'll learn about using Maven dependencies to load Camel modules and their third-party dependencies into your project.

Section 1.2 of chapter 1 has an overview of Apache Maven. If you need a Maven refresher, you might want to review that section before continuing here.

8.1.1 USING CAMEL MAVEN ARCHETYPES

Creating Maven-based projects is simple. You mainly have to worry about creating a POM file and the various standard directories that you'll use in your project. But if you're creating many projects, this can get repetitive because a lot of boilerplate setup is required for new projects.

Archetypes in Maven provide a means to define project templates and generate new projects based on those templates. They make creating new Maven-based projects easy because they create all the boilerplate POM elements, as well as key source and configuration files useful for particular situations.

NOTE For more information on Maven archetypes, see the guide: <http://books.sonatype.com/mvnref->

[book/reference/archetypes.html](#).

As illustrated in figure 8.1, this is all coordinated by the Maven archetype plugin. This plugin accepts user input and replaces portions of the archetype to form a new project.

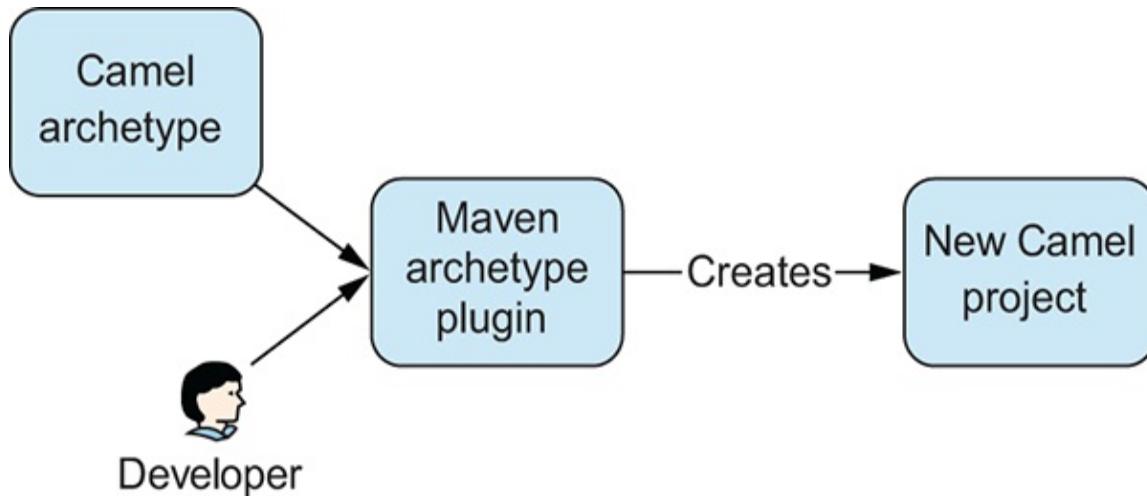


Figure 8.1 A Camel archetype and user input are processed by the Maven archetype plugin, which then creates a new Camel project.

To demonstrate how this works, let's look at the Maven quickstart archetype, which generates a plain Java application (no Camel dependencies). It's the default option when you run this command:

```
mvn archetype:generate
```

The archetype plugin asks you various questions, such as what `groupId` and `artifactId` to use for the generated project. When it's complete, you'll have a directory structure similar to this:

```
myApp
└── pom.xml
└── src
    ├── main
    │   └── java
    │       └── camelaction
    │           └── App.java
    └── test
        └── java
            └── camelaction
```

```
└— AppTest.java
```

In this structure, `myApp` is the `artifactId`, and `camelinaction` is the `groupId`. The archetype plugin created a `pom.xml` file, a Java source file, and a unit test, all in the proper locations.

NOTE Maven follows the paradigm of convention over configuration, so locations are important. Wikipedia provides details about this paradigm:
https://en.wikipedia.org/wiki/Convention_over_configuration.

Without any additional configuration, Maven knows that it should compile the Java source under the `src/main/java` directory and run all unit tests under the `src/test/java` directory. To kick off this process, you need to run the following Maven command:

```
mvn test
```

If you want to take it a step further, you could tell Maven to create a JAR file after compiling and testing by replacing the `test` goal with `package`.

You could start using Camel right from this example project, but it would involve adding Camel dependencies such as `camel-core`, starting up the `CamelContext`, and creating the routes. Although this wouldn't take that long, there's a much quicker solution: you can use one of the 11 archetypes provided by Camel to generate all this boilerplate Camel stuff for you. [Table 8.1](#) lists these archetypes and their main use cases.

Table 8.1 Camel's Maven archetypes

Archetype name	Description
<code>camel-archetype-blueprint</code>	Creates a Camel project that uses OSGi Blueprint. Ready to be deployed in OSGi.

camel-archetype-component	Creates a new Camel component.
camel-archetype-api-component	Creates a new Camel component that's based on a third-party API.
camel-archetype-cdi	Creates a Camel project with Contexts and Dependency Injection (CDI) support.
camel-archetype-connector	Creates a new Camel connector. A <i>connector</i> is a simplified and preconfigured component (set up for a specific use-case).
camel-archetype-dataformat	Creates a new Camel data format.
camel-archetype-java	Creates a Camel project that defines a sample route in the Java DSL.
camel-archetype-java8	Creates a Camel project that defines a sample route in the Java DSL using Java 8 features such as lambda expressions and method references.
camel-archetype-spring	Creates a Camel project that loads up a <code>CamelContext</code> in Spring and defines a sample route in the XML DSL.
camel-archetype-spring-boot	Creates a new Camel project by using Spring Boot. See the following note for an alternate way of creating Spring Boot-based Camel applications.
camel-archetype-web	Creates a Camel project that includes a few sample routes as a WAR file.

NOTE In addition to using the `camel-archetype-spring-boot` archetype to create Camel projects using Spring Boot, you can use the Spring Boot starter website at <http://start.spring.io>. This website allows you to customize the features/dependencies you

want to use in your project (Camel, web application, security, transactions, and so forth), so it greatly reduces project setup time. We covered Spring Boot in chapter 7, section 7.2.4.

Out of these 11 archetypes, the most commonly used one is probably the camel-archetype-java archetype. You'll try this next.

USING THE CAMEL-ARCHETYPE-JAVA ARCHETYPE

The camel-archetype-java archetype listed in table 8.1 boots up CamelContext and a Java DSL route. With this, we'll show you how to re-create the order-routing service for Rider Auto Parts as described in chapter 2. The project will be named `order-router`, and the package name in the source will be `camelinaction`.

To create the skeleton project for this service, run the following Maven command:

```
mvn archetype:generate \
  -B \
  -DarchetypeGroupId=org.apache.camel.archetypes \
  -DarchetypeArtifactId=camel-archetype-java \
  -DarchetypeVersion=2.20.1 \
  -DgroupId=camelinaction \
  -DartifactId=order-router
```

You specify the archetype to use by setting the `archetypeArtifactId` property to `camel-archetype-java`. You could replace this with any of the archetype names listed in table 8.1. The `archetypeversion` property is set to the version of Camel that you want to use.

The generate goal of the Maven archetype plugin can also be used in an interactive fashion. Just omit the `-B` option, and the plugin will prompt you through the archetype you want to use. You can select the Camel archetypes through this interactive shell as well, so it's a useful option for developers new to Camel.

After a few seconds of activity, Maven will have created an `order-router` subdirectory in the current directory. The `order-`

router directory layout is shown in the following listing.

Listing 8.1 Layout of the project created by camel-archetype-java

```
order-router
├── pom.xml
└── ReadMe.txt
└── src
    └── data
        └── message1.xml
```

Test data

```
    └── message2.xml
└── main
    └── java
        └── camelinaction
            └── MainApp.java
```

CamelContext setup

```
    └── MyRouteBuilder.java
```

Sample Java DSL route

```
    └── resources
        └── log4j.properties
```

Logging configuration

```
└── test
    └── java
        └── camelinaction
            └── resources
```

The archetype gives you a runnable Camel project, with a sample route and test data to drive it. The ReadMe.txt file tells you how to run this sample project: `run mvn compile exec:java`. Camel will continue to run until you press Ctrl-C, which causes Camel

to stop.

While running, the sample route consumes files in the `src/data` directory and, based on the content, routes them to one of two directories. If you look in the `target/messages` directory, you should see something like this:

```
target/messages
└── others
    └── message2.xml
└── uk
    └── message1.xml
```

Now you know that Camel is working on your system, so you can start editing `MyRouteBuilder.java` to look like the order-router application. You can begin by setting up FTP and web service endpoints that route to a JMS queue for incoming orders:

```
from("ftp://rider@localhost:21000/order?
password=secret&delete=true")
.to("jms:incomingOrders");

from("cxfrs:bean:orderEndpoint")
.inOnly("jms:incomingOrders")
.transform(constant("OK"));
```

At this point, if you try to run the application again using `mvn compile exec:java`, you'll get the following error message:

```
Failed to resolve endpoint:
ftp://rider@localhost:21000/order?
delete=true&password=secret due to: No component found with
scheme: ftp
```

Camel couldn't find the FTP component because it isn't on the classpath. You'd get the same error message for the CXF and JMS endpoints. There are other bits you have to add to your project to make this a runnable application: a test FTP server running on localhost, a CXF configuration, a JMS connection factory, and so on. A complete project is available in the book's source under `chapter8/order-router-full`.

For now, you'll focus on adding component dependencies using Maven.

8.1.2 USING MAVEN TO ADD CAMEL DEPENDENCIES

Technically, Camel is just a Java application. To use it, you add its JARs to your project's classpath. But using Maven to access these JARs will make your life a whole lot easier. Camel itself was developed using Maven for this very reason.

In the previous section, you saw that using an FTP endpoint with only the camel-core module as a dependency won't work. You need to add the camel-ftp module as a dependency to your project. In chapters 2 and 6, you saw that this was accomplished by adding the following to the dependencies section of the pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>2.20.1</version>
</dependency>
```

This dependency element tells Maven to download the camel-ftp JAR from Maven's central repository at <http://repo1.maven.org/maven2/org/apache/camel/camel-ftp/2.20.1/camel-ftp-2.20.1.jar>. This download URL is built up from Maven's central repository URL (<http://repo1.maven.org/maven2>) and Maven coordinates (`groupId`, `artifactId`, and so on) specified in the dependency element. After the download is complete, Maven will add the JAR to the project's classpath.

One detail that may not be obvious at first is that this dependency also has *transitive dependencies*. What are transitive dependencies? Well, in this case you have a project called order-router and you've added a dependency on camel-ftp. The camel-ftp module also has a dependency on commons-net, among others, so you can say that commons-net is a transitive dependency of order-router. Transitive dependencies are dependencies that a dependency has—the dependencies of the camel-ftp module, in this case.

When you add camel-ftp as a dependency, Maven will look up camel-ftp's POM file from the central Maven repository and look at the dependencies it has. Maven will then download and add those dependencies to this project's classpath.

The camel-ftp module adds a whopping 39 transitive dependencies to your project! Luckily, only 6 of them are needed at runtime; the other 33 are used during testing. The six transitive runtime dependencies can be viewed as a tree, as shown in [figure 8.2](#).

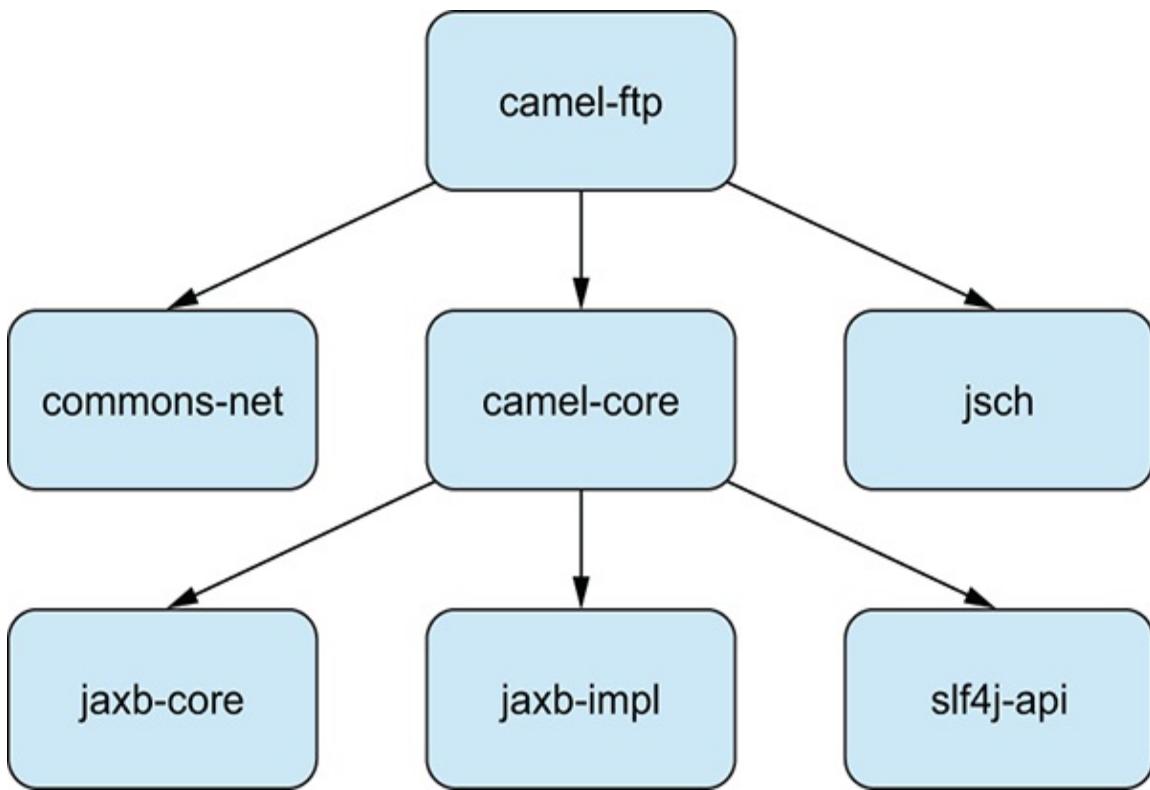


Figure 8.2 Transitive runtime dependencies of the camel-ftp module. When you add a dependency on camel-ftp to your project, you'll also get its transitive dependencies added to your classpath. In this case, commons-net, camel-core, and jsch are added. Additionally, camel-core has a dependency on jaxb-core, jaxb-impl, and slf4j-api, so these are added to the classpath as well.

You're already depending on camel-core in the order-router project, so only two dependencies—commons-net and jsch—are brought in by camel-ftp.

This is a view of only a small number of dependencies, but you can recognize that the dependency tree can get complex.

Fortunately, Maven finds these dependencies for you and resolves any duplicate dependencies. The bottom line is that when you're using Maven, you can worry less about your project's dependencies.

If you want to know what your project's dependencies are (including transitive ones), Maven offers the `dependency:tree` command. To see the dependencies in your project, run the following command:

```
mvn dependency:tree -Dscope=runtime
```

After a few seconds of work, Maven will print out a listing like this:

```
[INFO] --- maven-dependency-plugin:2.10:tree (default-cli)
@ chapter8-order-router ---
[INFO] camelinaction:chapter8-order-router:jar:2.0.0
[INFO] +- org.apache.camel:camel-core:jar:2.20.1:compile
[INFO] |  +- org.slf4j:slf4j-api:jar:1.7.21:compile
[INFO] |  +- com.sun.xml.bind:jaxb-core:jar:2.2.11:compile
[INFO] |  \- com.sun.xml.bind:jaxb-impl:jar:2.2.11:compile
[INFO] +- org.apache.camel:camel-spring:jar:2.20.1:compile
[INFO] |  +- org.springframework:spring-
core:jar:4.3.12.RELEASE:compile
[INFO] |  |  \- commons-logging:commons-
logging:jar:1.2:compile
[INFO] |  +- org.springframework:spring-
aop:jar:4.3.12.RELEASE:compile
[INFO] |  +- org.springframework:spring-
context:jar:4.3.12.RELEASE:compile
[INFO] |  +- org.springframework:spring-
beans:jar:4.3.12.RELEASE:compile
[INFO] |  +- org.springframework:spring-
expression:jar:4.3.12.RELEASE:compile
[INFO] |  \- org.springframework:spring-
tx:jar:4.3.12.RELEASE:compile
[INFO] +- org.apache.camel:camel-ftp:jar:2.20.1:compile
[INFO] |  +- com.jcraft:jsch:jar:0.1.54:compile
[INFO] |  \- commons-net:commons-net:jar:3.6:compile
[INFO] +- log4j:log4j:jar:1.2.17:compile
[INFO] \- org.slf4j:slf4j-log4j12:jar:1.7.21:compile
```

Here, you can see that Maven is adding 17 JARs to your project's runtime classpath, even though you added only `camel-core`,

camel-spring, camel-ftp, log4j, and slf4j-log4j12. Some dependencies are coming from several levels deep in the dependency tree.

Surviving without Maven

As you can imagine, adding all these dependencies to your project without the help of Maven would be tedious. If you absolutely must use an alternative build system, you can still use Maven to get the required dependencies for you by following these steps:

1. Download the POM file of the artifact you want. For camel-ftp, this would be
<http://repo1.maven.org/maven2/org/apache/camel/camel-ftp/2.20.1/camel-ftp-2.201.pom>.
2. Run `mvn -f camel-ftp-2.20.1.pom dependency:copy-dependencies`.
3. The dependencies for camel-ftp are located in the target/dependency directory. You can now use these in whatever build system you're using.

If you absolutely can't use Maven but would still like to use Maven repos, Apache Ivy (<http://ant.apache.org/ivy>) is a great dependency management framework for Apache Ant that can download from Maven repos. Other than that, you'll have to download the JARs yourself from the Maven central repo.

You now know all you need to develop Camel projects by using Maven. To make you an even more productive Camel developer, let's now look at developing Camel applications inside an IDE, such as Eclipse.

8.2 Using Camel in Eclipse

We haven't mentioned IDEs much so far, mostly because you don't need an IDE to use Camel. Certainly, though, you can't match the power and ease of use an IDE gives you. From a Camel point of view, having the Java or XML DSLs autocomplete for you makes route development a whole lot easier. The common Java debugging facilities and other tools will further improve your experience.

8.2.1 CREATING A NEW CAMEL PROJECT

Because Maven is used as the primary build tool for Camel projects, we'll show you how to use the Maven tooling in Eclipse to load up your Camel project. The standard Eclipse edition for Java developers has Maven tooling installed by default, but if you don't have that particular edition, you can easily search for and install the m2e plugin within the IDE. One thing that some developers like right away is that you don't have to leave the IDE to run Maven command-line tools during development. You can even access the Camel archetypes right from Eclipse. To demonstrate this feature, let's re-create the chapter8-order-router example you looked at previously.

Click File > New > Maven Project to start the New Maven Project wizard. Click Next on the first screen, and you'll be presented with a list of available archetypes, as shown in [figure 8.3](#). After filtering down to include only org.apache.camel, you can easily spot the Camel archetypes.

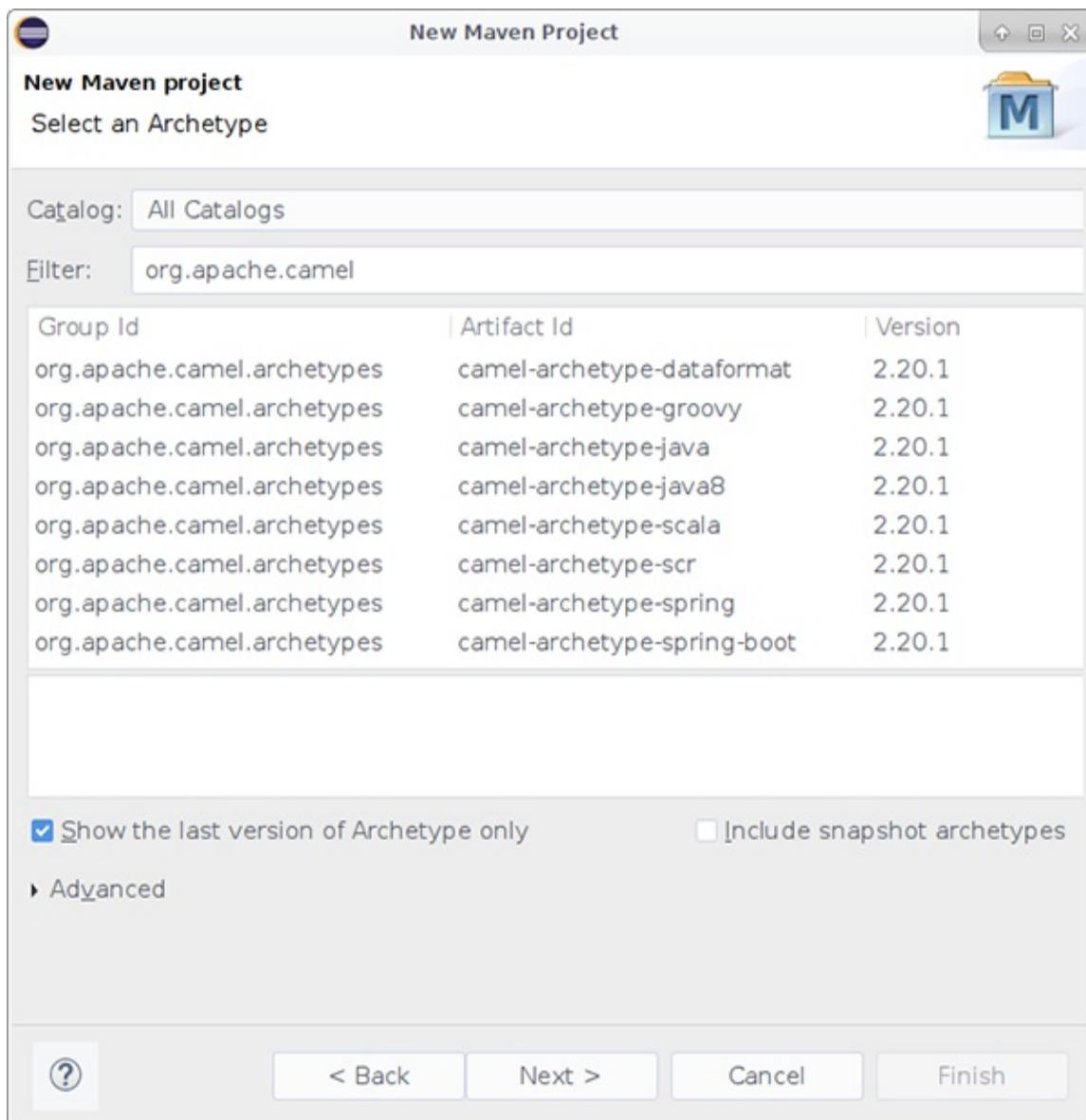


Figure 8.3 The New Maven Project wizard allows you to generate a new Camel project right in Eclipse.

Using Camel archetypes in this way is equivalent to the technique you used in section 8.1.1.

To run the order-router project with Eclipse, right-click the project in the Package Explorer and click Run As > Maven Build. This brings up an Edit Configuration dialog box where you can specify the Maven goals to use as well as any parameters. For the order-router project, use the exec:java goal, as shown in figure 8.4.

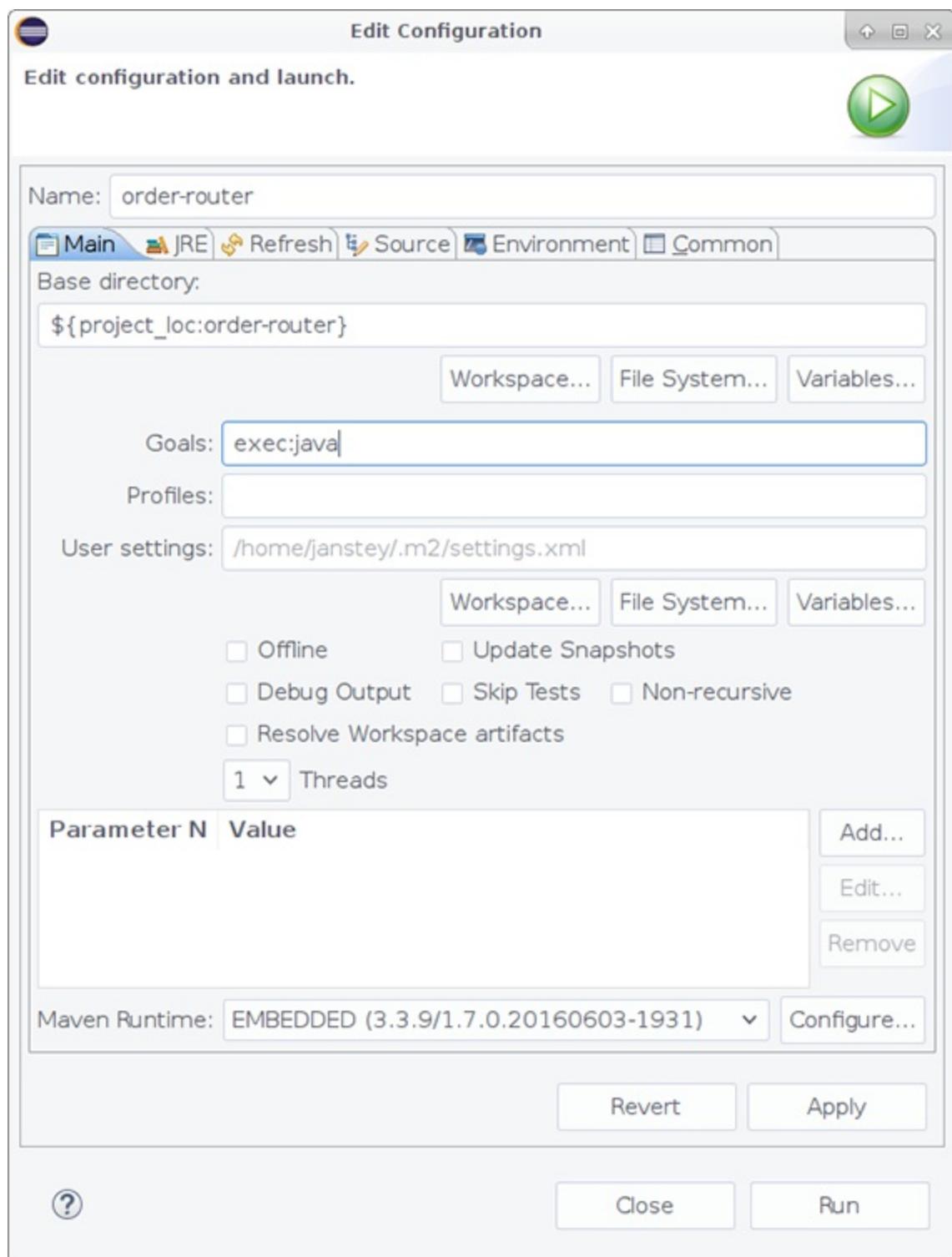
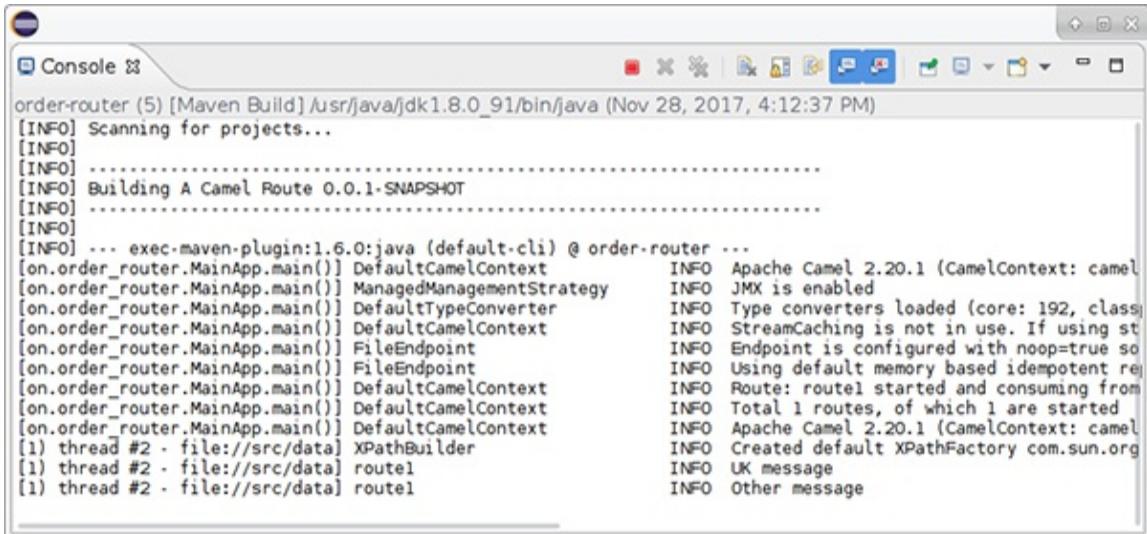


Figure 8.4 Right-clicking the order-router project in the Package Explorer and clicking Run As > Maven Build brings up this Edit Configuration dialog box. The exec:java Maven goal has been entered.

Clicking Run in this dialog box executes the mvn exec:java

command in Eclipse, with console output showing in the Eclipse Console view, as shown in [figure 8.5](#).



The screenshot shows the Eclipse IDE's Console view with the title "Console". The output is from a Maven build for the "order-router" project, specifically the "exec:java" goal. The log shows the application scanning for projects, building a Camel Route, and starting up Apache Camel 2.20.1. It also shows the creation of a DefaultCamelContext, the loading of type converters, and the configuration of StreamCaching. The route is identified as "routel" and consists of a single file endpoint. Camel is configured to use a noop=true strategy and is using default memory-based idempotent redelivery. The route is started, and a total of 1 route is reported. Camel 2.20.1 is used, and a default XPathFactory is created. The log ends with UK and other messages.

```
order-router (5) [Maven Build] /usr/java/jdk1.8.0_91/bin/java (Nov 28, 2017, 4:12:37 PM)
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building A Camel Route 0.0.1-SNAPSHOT
[INFO] -----
[INFO]
[INFO] ... exec-maven-plugin:1.6.0:java (default-cli) @ order-router ...
[on.order_router.MainApp.main()] DefaultCamelContext          INFO  Apache Camel 2.20.1 (CamelContext: camel
[on.order_router.MainApp.main()] ManagedManagementStrategy    INFO  JMX is enabled
[on.order_router.MainApp.main()] DefaultTypeConverter        INFO  Type converters loaded (core: 192, class
[on.order_router.MainApp.main()] DefaultCamelContext         INFO  StreamCaching is not in use. If using st
[on.order_router.MainApp.main()] FileEndpoint                 INFO  Endpoint is configured with noop=true so
[on.order_router.MainApp.main()] FileEndpoint                 INFO  Using default memory based idempotent re
[on.order_router.MainApp.main()] DefaultCamelContext         INFO  Route: routel started and consuming from
[on.order_router.MainApp.main()] DefaultCamelContext         INFO  Total 1 routes, of which 1 are started
[on.order_router.MainApp.main()] DefaultCamelContext         INFO  Apache Camel 2.20.1 (CamelContext: camel
[1] thread #2 - file://src/data] XPathBuilder                INFO  Created default XPathFactory com.sun.org
[1] thread #2 - file://src/data] routel                     INFO  UK message
[1] thread #2 - file://src/data] routel                     INFO  Other message
```

Figure 8.5 Console output from the order-router project when running the exec:java Maven goal

8.3 Debugging an issue with your new Camel project

You've just created a new Camel project in Eclipse by using an archetype, and now you want to know how to debug this thing if a problem arises. Well, Camel is technically just a Java framework, so you can set breakpoints wherever you like. But you'll hit a couple of issues if you try to step through your route in the debugger. Take the RouteBuilder from the order-router project, for example:

```
public class MyRouteBuilder extends RouteBuilder {
    public void configure() {
        from("file:src/data?noop=true")
            .choice()
                .when(xpath("/person/city = 'London'"))
                    .log("UK message")
                    .to("file:target/messages/uk")
                .otherwise()
                    .log("Other message")
                    .to("file:target/messages/others");
    }
}
```

```
}
```

If you try to set a breakpoint on, say, the `log("UK message")` line, you won't hit that breakpoint for each message satisfying `when(xpath("/person/city = 'London'"))`; the breakpoint will hit only once on startup. This is because RouteBuilders just forms the model from which Camel will create a graph of processors. The processor graph forms the runtime structure that you can debug on a per message basis. One trick that's quick to use in a pinch is an anonymous processor inside the route:

```
.when(xpath("/person/city = 'London'"))
    .log("UK message")
    .process(new Processor() {
        @Override
        public void process(Exchange exchange)
throws Exception {
            // breakpoint goes here!
        }
    })
    .to("file:target/messages/uk")
```

This isn't helpful to use in the XML DSL. Furthermore, having to modify your route just to debug isn't nice. What you can do is to not debug using the Eclipse debugger at all, but to use Camel's built-in management and monitoring abilities. In chapter 16, we discuss these topics:

- *JMX*—You may overlook JMX in your debugging toolbox for most applications, but Camel has an extensive set of data and operations exposed over JMX. Section 16.2 covers this in detail.
- *Logs*—Logs are the first place you should go when investigating a problem in most applications, and Camel is no different. Section 16.3 covers this.
- *Tracing*—The tracer is probably Camel's most useful built-in tool to diagnose problems. Enabling tracing will log each message at every step of a route. In this way, you can see where a message goes through a route and how it changes at each step. Section 16.3.4 covers this in detail.

If you want to go beyond what Camel has to offer, several tooling projects are out there to assist you with debugging Camel.

Chapter 19 covers the following:

- *JBoss Fuse Tooling*—The Fuse Tooling is an Eclipse plugin for Camel development that includes, among many other things, a visual debugger for Camel routes. Section 19.1.1 covers this.
- *hawtio*—The hawtio project is a highly extensible web console for managing Java applications. It has a nice Camel plugin that allows you to step through a route visually in your browser!

DEBUGGING YOUR UNIT TESTS

Tooling projects such as hawtio and JBoss Fuse Tooling both tie into the same Camel debugging support to achieve visual debugging. The main debugger API is

`org.apache.camel.spi.Debugger`, which contains methods to set breakpoints in routes, step through a route, and so forth. The default implementation is

`org.apache.camel.impl.DefaultDebugger`, which is in the camel-core module. How does this help you? Well, most of the Camel testing modules have support to use this default debugger implementation in your unit tests. You can see what's happening before and after each processor invocation in a Camel route's runtime processor graph. [Table 8.2](#) lists the Maven modules as well as which test class to extend to gain access to the debugger.

Table 8.2 Test modules that support the Camel debugger

Maven module	Test support class
camel-test	<code>org.apache.camel.test.junit4.CamelTestSupport</code>
camel-test-spring	<code>org.apache.camel.test.spring.CamelSpringTestSupport</code>
camel-test-blueprint	<code>org.apache.camel.test.blueprint.CamelBlueprintTestSupport</code>

blueprint	
camel-test-karaf	org.apache.camel.test.karaf.CamelKarafTestSupport

After you've extended one of the compatible test support classes, you have to override `isUseDebugger` as follows:

```
@Override
public boolean isUseDebugger() {
    return true;
}
```

Next, you have to override one or both of the methods invoked around processor invocations:

```
@Override
protected void debugBefore(Exchange exchange, Processor
processor,
    ProcessorDefinition<?> definition, String id,
String shortName) {
    log.info("MyDebugger: before " + definition + " with
body "
        + exchange.getIn().getBody());
}

@Override
protected void debugAfter(Exchange exchange, Processor
processor,
    ProcessorDefinition<?> definition, String id,
String label,
    long timeTaken) {
    log.info("MyDebugger: after " + definition + " took " +
timeTaken
        + " ms, with body " + exchange.getIn().getBody());
}
```

Now when you run your unit test, `debugBefore` will be invoked before each processor, and `debugAfter` will be invoked after. If you set a breakpoint inside each of these methods, you can inspect what's happening to Exchange precisely. You can also run the test case, and these debug methods will provide detailed tracing similar to Camel's tracer.

To try this for yourself, you can run the example in the chapter8/order-router-full directory:

```
mvn clean test -Dtest=OrderRouterDebuggerTest
```

Now you can say that you know how to debug a Camel application! Let's next discuss how to start extending Camel itself by adding your own components.

8.4 Developing custom components

For most integration scenarios, a Camel component is available to help. Sometimes, though, no Camel component is available, and you need to bridge Camel to another transport, API, data format, and so on. You can do this by creating your own custom component.

Creating a Camel component is relatively easy, which may be one of the reasons that custom Camel components frequently show up on other community sites, in addition to the official Camel distribution. In this section, you'll create your own custom component for Camel.

8.4.1 SETTING UP A NEW CAMEL COMPONENT

One of the first things to do when developing a custom component is to decide what endpoint name to use. This name will be used to reference the custom component in an endpoint URI. You need to make sure this name doesn't conflict with a component that already exists by checking the online component list (<http://camel.apache.org/components.html>). As with a regular Camel project, you can start creating a new component by using a Maven archetype to generate a skeleton project. To create a new Camel component with camelinaction.component as the package name, custom as the artifactId, MyComponent as the component name, and mycomponent as the endpoint name, run the following Maven command:

```
mvn archetype:generate \
-B \
```

```
-DarchetypeGroupId=org.apache.camel.archetypes \
-DarchetypeArtifactId=camel-archetype-component \
-DarchetypeVersion=2.20.1 \
-DgroupId=camelinaction.component \
-DartifactId=custom \
-Dname=My \
-Dscheme=mycomponent \
-Dversion=1.0-SNAPSHOT
```

This generates a project structure like that shown in the following listing.

Listing 8.2 Layout of a project created by camel-archetype-component

```
custom
└── pom.xml
└── ReadMe.txt
└── src
    ├── data
    └── main
        └── java
            └── camelinaction
                └── component
                    └── MyComponent.java
```

❶

Component implementation

```
└── resources
    └── META-INF
        └── services
            └── org
                └── apache
                    └── camel
                        └── component
                            └── mycomponent
                                └── MyConsumer.java      ❶
                                └── MyEndpoint.java     ❶
                                └── MyProducer.java     ❶
```

❷

File that maps URI scheme to component class

```
└── test
    └── java
```

```
└── camelinaction  
    └── component  
        └── MyComponentTest.java
```

3

Test case for component

```
└── resources  
    └── log4j.properties
```

This is a fully functional Hello World demo component containing a simple consumer that generates dummy messages at regular intervals, and a producer that prints a message to the console. You can run the test case included with this sample component by running the following Maven command:

```
mvn test
```

This project is also available in the chapter8/custom directory of the book's source.

Your component can now be used in a Camel endpoint URI. But you shouldn't stop here. To understand how these classes make up a functioning component, you need to understand the implementation details of each.

8.4.2 DIVING INTO THE IMPLEMENTATION

The four classes that make up a component in Camel have been mentioned several times before. To recap, it all starts with the Component class, which then creates an Endpoint. An Endpoint, in turn, can create producers and consumers. This is illustrated in figure 8.6.

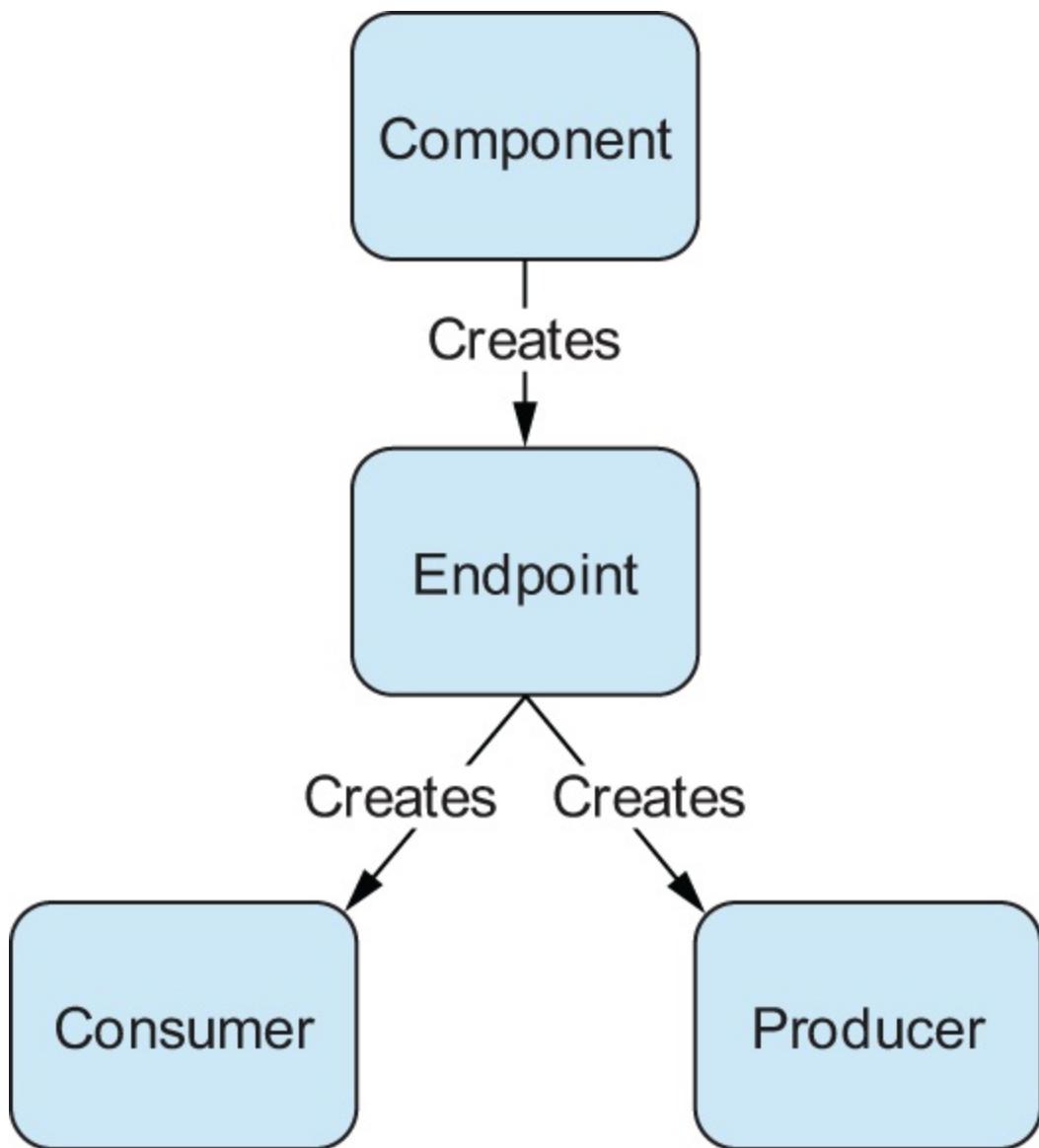


Figure 8.6 A component creates an endpoint, which then creates producers and consumers.

You'll first look into the component and Endpoint implementations of the custom `MyComponent` component.

COMPONENT AND ENDPOINT CLASSES

The first entry point into a Camel component is the class implementing the `Component` interface. A component's main job is to be a factory of new endpoints. It does a bit more than this under the hood, but typically you don't have to worry about these details because they're contained in the super class.

The `DefaultComponent` allows you to define the configurable parameters of the endpoint URI via annotations, which we'll touch on when you look at the implementation of `MyEndpoint`. The `MyComponent` class generated by the camel-archetype-component archetype forms a simple and typical component class structure, as shown here:

```
package camelinaction.component;
import java.util.Map;
import org.apache.camel.CamelContext;
import org.apache.camel.Endpoint;
import org.apache.camel.impl.DefaultComponent;
public class MyComponent extends DefaultComponent {
    protected Endpoint createEndpoint(String uri, String
remaining,
                                     Map<String, Object>
parameters) throws Exception {
    Endpoint endpoint = new MyEndpoint(uri, this);
    setProperties(endpoint, parameters);
    return endpoint;
}
}
```

This class is straightforward, except perhaps for the way in which properties are set with the `setProperties` method. This method takes in the properties set in the endpoint URI string, and for each will invoke a setter method on the endpoint through reflection. For instance, say you used the following endpoint URI:

```
mycomponent:endpointName?prop1=value1&prop2=value2
```

The `setProperties` method, in this case, would try to invoke `setProp1("value1")` and `setProp2("value2")` on the endpoint. Camel will take care of converting those values to the appropriate type.

The endpoint itself is also a relatively simple class, as shown in the following listing.

Listing 8.3 Custom Camel endpoint—`MyEndpoint`

```
package camelinaction.component;
```

```
import org.apache.camel.Consumer;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultEndpoint;
import org.apache.camel.spi.Metadata;
import org.apache.camel.spi.UriEndpoint;
import org.apache.camel.spi.UriParam;
import org.apache.camel.spi.UriPath;
```

@UriEndpoint(1

1

Specifies that this endpoint is set up with annotations

```
firstVersion = "1.0-SNAPSHOT",
scheme = "mycomponent",
title = "Custom",
syntax="mycomponent:name",
consumerClass = MyConsumer.class,
label = "custom")
public class MyEndpoint extends DefaultEndpoint {      2
```

2

Extends from default endpoint class

@UriPath @Metadata(required = "true") 3

3

Specifies the URI path content

```
private String name;
@UriParam(defaultValue = "10")      4
```

4

Specifies that this is a settable option on the endpoint URI

```
private int option = 10;

public MyEndpoint() {
}
```

```
public MyEndpoint(String uri, MyComponent component) {  
    super(uri, component);  
}  
  
public MyEndpoint(String endpointUri) {  
    super(endpointUri);  
}  
  
public Producer createProducer() throws Exception {  
    return new MyProducer(this);      5  
}
```

5

Creates new producer

```
}
```

```
public Consumer createConsumer(Processor processor)  
throws Exception {  
    return new MyConsumer(this, processor);      6  
}
```

6

Creates new consumer

```
}
```

```
public boolean isSingleton() {  
    return true;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setOption(int option) {  
    this.option = option;  
}  
  
public int getOption() {  
    return option;  
}
```

```
}
```

The first thing you'll notice is that a lot of setup is handled via annotations on the class and fields. What won't be obvious is that these are mainly used to generate documentation for the Camel components using `camel-package-maven-plugin`. This is also used by tooling (as discussed in chapter 19) to better determine configurable options, URI syntax, grouping of component by labels, and so forth. If you want your component to be widely reused, it's important to add these annotations. But they're not strictly required.

NOTE You can find more information on Camel's various endpoint annotations on the Camel website:
<http://camel.apache.org/endpoint-annotations.html>.

The first annotation used, `@UriEndpoint`, tells Camel to expect that this endpoint is described via annotations ①. Here you can specify things such as the scheme of the endpoint, its syntax, and any labels that you want to add. Labels are most often used to categorize things in Camel. For example, if this component integrated with a messaging system, you could add the label `messaging`.

As with the `MyComponent` class, you're deriving from a default implementation class from `camel-core` here too ②. In this case, you're extending the `DefaultEndpoint` class. It's common when creating a new Camel component to have the `Component`, `Endpoint`, `Consumer`, and `Producer` all derive from default implementations in `camel-core`. This isn't necessary, but it makes new component development much easier, and you always benefit from the latest improvements to the default implementations without having to code them yourself.

Moving on to the class fields, you have a few more uses of annotations. `@UriPath` specifies the part of the URI between the scheme and the options ③. `Metadata` is used to add extra info

about an endpoint option, such as whether it's required or has a label associated with it ❸. `@UriParam` specifies that this is a settable option on the endpoint URI. That means with the code in [listing 8.3](#), you can have an option like this:

```
mycomponent : endpointName?option=value1
```

As mentioned in chapter 6, the `Endpoint` class acts as a factory for both consumers and producers. In this example, you're creating both producers ❹ and consumers ❺, which means that this endpoint can be used in a `to` or `from` Java DSL method. Sometimes you may need to create a component that has only a producer or consumer, not both. In that case, it's recommended that you throw an exception, so users know it isn't supported:

```
public Producer createProducer() throws Exception {  
    throw new UnsupportedOperationException(  
        "You cannot send messages to this endpoint: " +  
        getEndpointUri());  
}
```

The real bulk of most components is in the producer and consumer. The `Component` and `Endpoint` classes are mostly designed to fit the component into Camel. In the producers and consumers, which we'll look at next, you have to interface with the remote APIs or marshal data to a particular transport.

PRODUCERS AND CONSUMERS

The producer and consumer are where you get to implement how messages will get on or off a particular transport—in effect, bridging Camel to something else. This is illustrated in [figure 8.7](#).

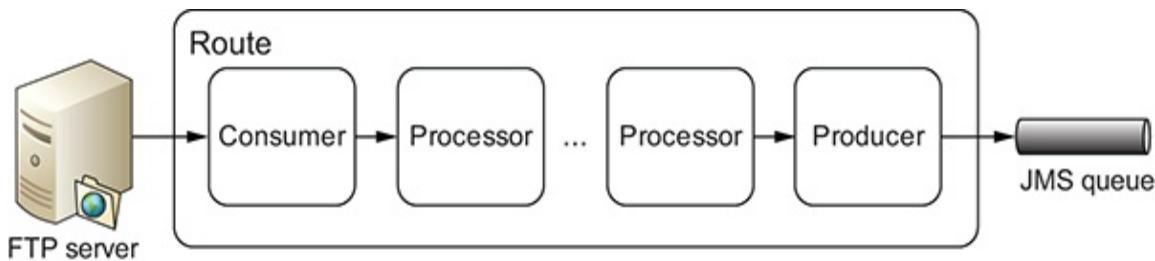


Figure 8.7 A simplified view of a route in which the consumer and producer handle interfacing with external systems. Consumers take messages from an external system into Camel, and producers send messages to external systems.

In your skeleton component project that was generated from an archetype, a producer and consumer are implemented and ready to go. These were instantiated by the `MyEndpoint` class in [listing 8.3](#). The producer, named `MyProducer`, is shown in the following listing.

Listing 8.4 Custom Camel producer—`MyProducer`

```
package camelaction.component;

import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultProducer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MyProducer extends DefaultProducer { 1

1
Extends from default producer class

    private static final Logger LOG =
LoggerFactory.getLogger(MyProducer.class);
    private MyEndpoint endpoint;

    public MyProducer(MyEndpoint endpoint) {
        super(endpoint);
        this.endpoint = endpoint;
    }

    public void process(Exchange exchange) throws Exception
{ 2

2
Serves as entry point to producer

    System.out.println(exchange.getIn().getBody()); 3
```

3

Prints message body

```
}
```

```
}
```

Like the `Component` and `Endpoint` classes, the producer also extends from a default implementation class from camel-core called `DefaultProducer` ①. The `Producer` interface extends from the `Processor` interface, so you use a process method ②. As you can probably guess, a producer is called in the same way as a processor, so the entry point into the producer is the `process` method. The sample component that was created automatically has a basic producer—it just prints the body of the incoming message to the screen ③. If you were sending to an external system instead of the screen, you'd have to handle a lot more here, such as connecting to a remote system and marshaling data. In the case of data marshaling, it's often a good idea to implement this by using a custom `TypeConverter`, as described in chapter 3, which makes the converters available to other parts of your Camel application.

You can see how messages could be sent out of a route, but how do they get into a route? Consumers, like the `MyConsumer` class generated in your custom component project, get the messages into a route. The `MyConsumer` class is shown in the following listing.

Listing 8.5 Custom Camel consumer—`MyConsumer`

```
package camelinaction.component;

import java.util.Date;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.impl.ScheduledPollConsumer;

public class MyConsumer extends ScheduledPollConsumer
```

```
{ ①
```

①

Extends from built-in consumer

```
    private final MyEndpoint endpoint;  
  
    public MyConsumer(MyEndpoint endpoint, Processor  
processor) {  
        super(endpoint, processor);  
        this.endpoint = endpoint;  
    }  
  
    @Override  
    protected int poll() throws Exception { ②
```

②

Is called every 500 ms

```
        Exchange exchange = endpoint.createExchange();  
  
        // create a message body  
        Date now = new Date();  
        exchange.getIn().setBody("Hello World! The time is  
" + now);  
  
        try {  
            getProcessor().process(exchange); ①
```

③

Sends to next processor in route

```
            return 1; // number of messages polled  
        } finally {  
            // log exception if an exception occurred and  
was not handled  
            if (exchange.getException() != null) {  
  
getExceptionHandler().handleException("Error processing  
exchange", exchange, exchange.getException());  
            }  
        }
```

}

The consumer interface itself doesn't impose many restrictions or give any guidelines as to how a consumer should behave, but the `DefaultConsumer` class does, so it's helpful to extend from this class when implementing your own consumer. In [listing 8.5](#), you extend from a subclass of `DefaultConsumer`, the `ScheduledPollConsumer` **1**. This consumer has a timer thread that will invoke the `poll` method every 500 milliseconds **2**.

TIP See the discussion of the Scheduler and Quartz components in chapter 6 for more information on creating routes that need to operate on a schedule.

Typically, a consumer will either poll a resource for a message or set up an event-driven structure for accepting messages from remote sources. In this example, you have no remote resource, so you can create an empty exchange and populate it with a Hello World message. A real consumer still would need to do this.

A common pattern for consumers is something like this:

```
Exchange exchange = endpoint.createExchange();
// populate exchange with data
getProcessor().process(exchange);
```

Here you create an empty exchange, populate it with data, and send it to the next processor in the route **3**.

At this point, you should have a good understanding of what's required to create a new Camel component. You may even have a few ideas about what you'd like to bridge Camel to next!

8.5 Generating components with the API component framework

Now you know how to write a component from a Camel API

point of view. But what do you do when you start interacting with other transports or APIs? Often the first major decision you have to make is whether you'll have to code all these interactions from scratch or if there's already a suitable library out there that you can reuse. Components in Camel's core module take the first approach if you're looking for inspiration for your implementation. Most of the other component modules take the latter approach and are backed by a third-party library.

Using a third-party library significantly reduces the size and complexity of the component because most of the grunt work is handled in the third-party library. Component development in this case becomes a mundane task. You saw the boilerplate code from a typical component in the previous section; now your task becomes mapping elements of an endpoint URI to method calls in this third-party library.

For larger libraries, this process can be daunting. Take, for example, the camel-box component, which integrates with the box.com file-management service. This component has more than 50 operations accessible from the endpoint URI. That would have been a pain to implement. Dhiraj Bokde, a colleague of your authors, realized we could do so much better. Before drudging through 50-plus operations for the box component, he created tooling called the *API component framework* to generate much of this mapping code. Let's take a look at how this can be done.

8.5.1 GENERATING THE SKELETON API PROJECT

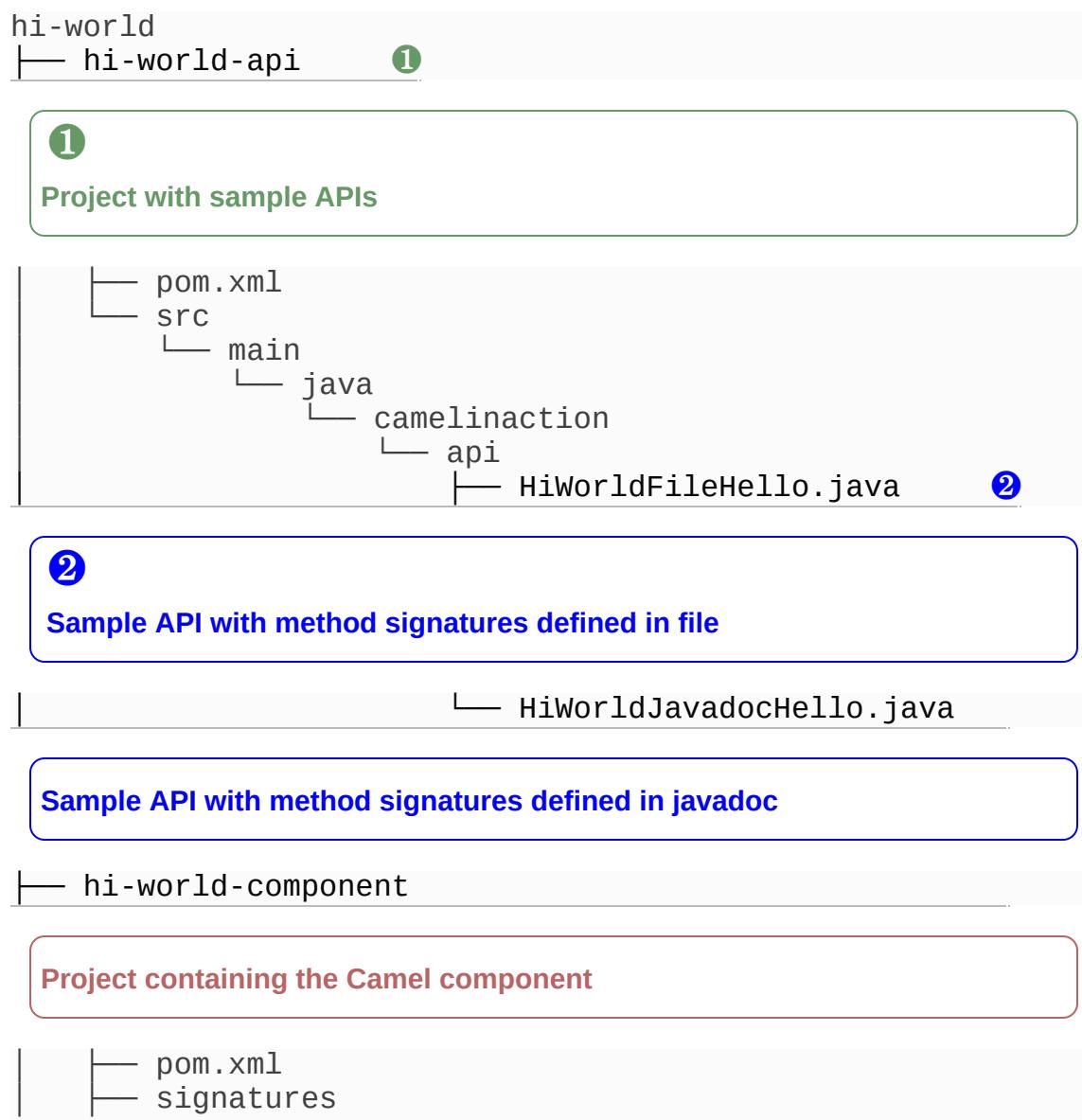
To generate a skeleton project using the Camel API component framework, you need to use the camel-archetype-api-component archetype. Try this with the following Maven command:

```
mvn archetype:generate \
-B \
-DarchetypeGroupId=org.apache.camel.archetypes \
-DarchetypeArtifactId=camel-archetype-api-component \
-DarchetypeVersion=2.20.1 \
```

```
-DgroupId=camelinaction \
-DartifactId=hi-world \
-Dname=HiWorld \
-Dscheme=hiworld
```

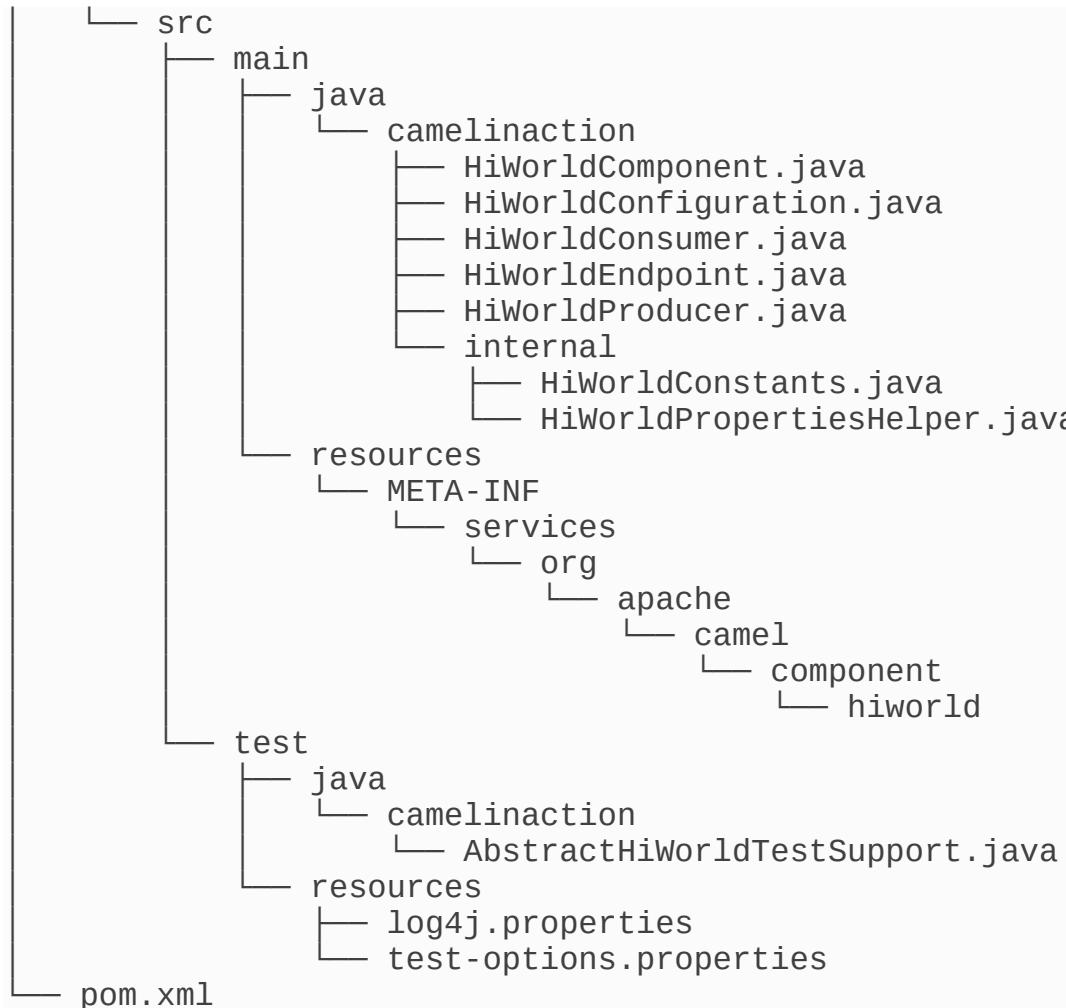
The `name` property is used in things such as class names (for example, `HiWorldComponent`), and `scheme` is used to form endpoint URIs (for example, `to("hiworld://...")`). The resultant `hi-world` directory layout is shown in the following listing.

Listing 8.6 Layout of the project created by camel-archetype-api-component



```
|   |   └ file-sig-api.txt ②
```

Method signatures for API ②



The generated multimodule Maven project is a bit bigger than the custom component developed in the previous section. This is because it contains a lot of sample code for you to play with. If you were creating a real API component, you'd most likely need to keep only a portion of the hi-world-component project ②.

The hi-world-api project ① is the sample third-party API that your Camel component ② will be using. In a real Camel API component, instead of this sample API project, you'd be pointing the tooling at an already released library. For example, in the camel-box component (part of Apache Camel), a dependency is

on a third-party library:

```
<dependency>
  <groupId>net.box</groupId>
  <artifactId>boxjavalibv2</artifactId>
</dependency>
```

Instead, your project will depend on the sample API project:

```
<dependency>
  <groupId>camelinaaction</groupId>
  <artifactId>hi-world-api</artifactId>
</dependency>
```

Let's look at how to generate a working Camel component from this API.

8.5.2 CONFIGURING THE CAMEL-API-COMPONENT-MAVEN-PLUGIN

In [listing 8.6](#), you saw that two projects were listed: a sample API and a Camel component using that sample API. Looking at the pom.xml for the Camel component, you first have a dependency on the sample API, as follows:

```
<dependency>
  <groupId>camelinaaction</groupId>
  <artifactId>hi-world-api</artifactId>
</dependency>
<dependency>
  <groupId>camelinaaction</groupId>
  <artifactId>hi-world-api</artifactId>
  <classifier>javadoc</classifier>
  <scope>provided</scope>
</dependency>
```

Now, you may be wondering why we included a Javadoc JAR in addition to the main JAR. It comes down to a lack of information when interrogating the API JAR via reflection. Method signatures are maintained, but parameter names aren't. In order to get the full set of names (method and parameter names), you have to fill in the blanks for the tooling with either a signature file or Javadoc. Fortunately, Javadoc is a commonly available

resource for Java APIs, so this will likely be the easiest option. The following listing shows the minimal camel-api-component-maven-plugin configuration for using both a signature file and Javadoc to supply parameter names.

Listing 8.7 Minimal configuration for the camel-api-component-maven-plugin

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-api-component-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>generate-test-component-classes</id>
      <goals>
        <goal>fromApis</goal>
      </goals>
      <configuration>
        <apis>
          <api>
            <apiName>hello-file</apiName> ①

```

1

API name for file-based signature

```
<proxyClass>camelinaction.api.HiWorldFileHello</proxyClass>
```

Class to interrogate using reflection

```
  <fromSignatureFile>signatures/file-sign-
  api.txt</fromSignatureFile> ③
```

3

Location of text file with method signatures for class

```
  </api>
  <api>
    <apiName>hello-javadoc</apiName> ②
```

2

API name for Javadoc-based signature

```
<proxyClass>camelinaction.api.HiWorldJavadocHello</proxyClass>
```

Class to interrogate using reflection

```
<fromJavadoc/> 4
```

4

Specify to check for method signatures in javadoc for class defined in the preceding line

```
        </api>
        </apis>
    </configuration>
    </execution>
    </executions>
</plugin>
```

Here you can see that the Maven plugin is configured to generate an endpoint URI to the Java API mapping for two classes ① ②. Let's first take a look at the file-supplemented API `camelinaction.api.HiWorldFileHello`, shown in the following listing.

Listing 8.8 Sample HiWorld API

```
package camelinaction.api;

/**
 * Sample API used by HiWorld Component whose method
signatures are read from File.
 */
public class HiWorldFileHello {

    public String sayHi() {
        return "Hello!";
}
```

```
public String greetMe(String name) {
    return "Hello " + name;
}

public String greetUs(String name1, String name2) {
    return "Hello " + name1 + ", " + name2;
}
}
```

When the Maven plugin interrogates `HiWorldFileHello`, it finds three methods and determines that three sub-URI schemes will be needed:

```
hiworld://hello-file/greetMe?inBody=name
hiworld://hello-file/greetUs
hiworld://hello-file/sayHi
```

The `hiworld` portion of the URI comes from the `-Dscheme=hiworld` argument passed into the Maven archetype plugin in section 8.4.1. The `hello-file` portion of the URI comes from **①** in [listing 8.7](#). The last three parts (`greetMe`, `greetUs`, and `sayHi`) come from the methods in the `HiWorldFileHello` class.

Input data is handled differently, depending on the number of parameters. For methods with just one parameter, the full message body is often used as the data. To set this, you use the `inBody` URI option and specify the parameter name you want the message body to map to. A URI like this

```
hiworld://hello-file/greetMe?inBody=name
```

will be calling the following:

```
HiWorldFileHello.greetMe(/* Camel message body */)
```

For methods with more than one parameter, each parameter is mapped to a message header. For the `greetUs` method, you'd set up headers like this:

```
final Map<String, Object> headers = new HashMap<String, Object>();
headers.put("CamelHiWorld.name1", /* Data for name1 parameter here */);
```

```
headers.put("CamelHiWorld.name2", /* Data for name2  
parameter here */);
```

The header names are generated as follows:

- Camel is always used for the prefix.
- The HiWorld portion of the URI comes from the -Dname=HiWorld argument passed into the Maven archetype plugin in section 8.4.1.
- Finally, there's a dot followed by the parameter name.

Recall that for this API, you're using a signature file to provide the parameter names. We specified this in [listing 8.7 ③](#). The simple signature file signatures/file-sig-api.txt is shown here:

```
public String sayHi();  
public String greetMe(String name);  
public String greetUs(String name1, String name2);
```

Without this file, you'd have only parameter names such as arg0, arg1, and so forth.

This process is even easier when you have Javadoc available for the third-party API. You tried the Javadoc way of providing parameter names in [listing 8.7 ④](#). The proxy class that you're using, camelaction.api.HiWorldJavadocHello, is essentially the same as camelaction.api.HiWorldFileHello shown in [listing 8.8](#). It produces three sub-URI schemes like this:

```
hiworld://hello-javadoc/greetMe?inBody=name  
hiworld://hello-javadoc/greetUs  
hiworld://hello-javadoc/sayHi
```

These endpoints are used in the same way as the file-supplemented ones from before.

For these simple handcrafted APIs, you don't need to change much in the conversion to Camel endpoints. For larger APIs, though, customization is a likely requirement. A ton of customization options are available to address this, so let's go over them.

8.5.3 SETTING ADVANCED CONFIGURATION OPTIONS

At times you may need to customize the mapping from a third-party API to a Camel endpoint URI. For instance, some API methods may not make sense to call on their own, or maybe you want to keep the scope of your component in check. Perhaps the naming used on the remote API also has conflicts with other names you want to use in your component. Or perhaps you think you can name things better.

Let's look at some of the advanced configuration options you have available for both file- and Javadoc-supplemented API components.

GLOBAL CONFIGURATION OPTIONS

You can apply six configuration options to both file- and javadoc-supplemented API components as well:

- **substitutions**—Substitutions are used to modify parameter names. This could be useful to avoid name clashes or if you feel you can provide a more descriptive name for your Camel component versus what was provided in the third-party API. For example, in our API the `greetMe` method has a single parameter called `name`. Perhaps it would be more useful in your endpoint URI to have this as `username` if it corresponds to a system `username`:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-api-component-maven-
plugin</artifactId>
  <executions>
    <execution>
      <id>generate-test-component-classes</id>
      <goals>
        <goal>fromApis</goal>
      </goals>
      <configuration>
        <apis>
          ...
        </apis>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```

    </apis>
    <substitutions>
        <substitution>
            <method>^greetMe$</method>
            <argName>^( .+)$</argName>
            <replacement>user$1</replacement>
        </substitution>
    </substitutions>
</configuration>
</execution>
</executions>
</plugin>

```

The `method` and `argName` elements are regular expressions. You can refer to regular expression group matches in the `replacement` element as `$1`, `$2`, and so forth. You can also use a regular expression to grab text out of the parameter type. To do this, you set the `replaceWithType` element to `true` and specify a regular expression in the `argType` element. The following example changes the `greetMe` method's `username` parameter to `usernameString`:

```

<substitutions>
    <substitution>
        <method>^greetMe$</method>
        <argName>^name$</argName>
        <argType>^java.lang.( .+)$</argType>
        <replacement>username$1</replacement>
        <replaceWithType>true</replaceWithType>
    </substitution>
</substitutions>

```

- **aliases**—Method aliases are useful for creating shorthand endpoint URI option names for long-winded method parameter names. For example, let's provide shorter names for your `greetMe/us` methods:

```

<plugin>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-api-component-maven-
plugin</artifactId>
    <executions>
        <execution>
            <id>generate-test-component-classes</id>
            <goals>
                <goal>fromApis</goal>

```

```
</goals>
<configuration>
  <apis>
    ...
  </apis>
  <aliases>
    <alias>
      <methodPattern>greet(.+)</methodPattern>
      <methodAlias>$1</methodAlias>
    </alias>
  </aliases>
</configuration>
</execution>
</executions>
</plugin>
```

The `methodPattern` element is a regular expression that matches method names. Any groups captured from the method names are available in the element as `$1`, `$2`, and so on. In this case, you've made the following two URIs equivalent:

```
hiworld://hello-javadoc/greetUs
hiworld://hello-javadoc/us
```

And you've made the following two equivalent:

```
hiworld://hello-javadoc/greetMe?inBody=name
hiworld://hello-javadoc/me?inBody=name
```

- `nullableOptions`—By specifying parameters as nullable, if you neglect to set a corresponding header or message body, a null will be passed into the third-party API. It's useful to be able to do this because null can be a valid parameter value. Take the following example:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-api-component-maven-
plugin</artifactId>
  <executions>
    <execution>
      <id>generate-test-component-classes</id>
      <goals>
        <goal>fromApis</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```
<configuration>
  <apis>
    ...
  </apis>
  <nullableOptions>
    <nullableOption>name</nullableOption>
  </nullableOptions>
</configuration>
</execution>
</executions>
</plugin>
```

With this configuration, any `name` parameters will be allowed to be null.

- `excludeConfigNames`—Exclude any parameters with a name matching the specified regular expression.
- `excludeConfigTypes`—Exclude any parameters with a type matching the specified regular expression.
- `extraOptions`—Add extra options to your endpoint URI that weren't in the third-party API. For example, let's add a `languageoption`:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-api-component-maven-
plugin</artifactId>
  <executions>
    <execution>
      <id>generate-test-component-classes</id>
      <goals>
        <goal>fromApis</goal>
      </goals>
      <configuration>
        <apis>
          ...
        </apis>
        <extraOptions>
          <extraOption>
            <type>java.lang.String</type>
            <name>language</name>
          </extraOption>
        </extraOptions>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```
</execution>
</executions>
</plugin>
```

JAVADOC-ONLY CONFIGURATION OPTIONS

Some additional options are available only within the `fromJavadoc` element. Let's take a look at them:

- `excludePackages`—Exclude methods from proxy classes with a specified package name. Useful for filtering out methods from unwanted superclasses. Methods from any classes in the `java.lang` package are excluded by default:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-api-component-maven-
  plugin</artifactId>
  <executions>
    <execution>
      <id>generate-test-component-classes</id>
      <goals>
        <goal>fromApis</goal>
      </goals>
      <configuration>
        <apis>
          <api>
            <apiName>hello-javadoc</apiName>

<proxyClass>camelinaction.api.HiWorldJavadocHello
            </proxyClass>
            <fromJavadoc>

<excludePackages>package.name.to.exclude</excludePackages>
            </fromJavadoc>
          </api>
        </apis>
      </configuration>
    </execution>
  </executions>
</plugin>
```

- `excludeClasses`—Exclude classes (unwanted superclasses) matching a specified name:

```
<fromJavadoc>
```

```
<excludeClasses>SomeUnwantedAbstractSuperClass</excludeClasses>
</fromJavadoc>
```

- `includeMethods`—Only methods matching this pattern will be included:

```
<fromJavadoc>
  <includeMethods>greetMe</includeMethods>
</fromJavadoc>
```

Include only the `greetMe` method in processing.

- `excludeMethods`—Any methods matching this pattern won't be included:

```
<fromJavadoc>
  <excludeMethods>greetMe</excludeMethods>
</fromJavadoc>
```

Don't include the `greetMe` method in processing.

- `includeStaticMethods`—Also include static methods. This is `false` by default:

```
<fromJavadoc>
  <includeStaticMethods>true</includeStaticMethods>
</fromJavadoc>
```

8.5.4 IMPLEMENTING REMAINING FUNCTIONALITY

With the sample component generated from `camel-archetype-api-component`, there isn't much left to do. But if you were running this against a real third-party API, you'd certainly need to fill in some blanks.

SETTING UP UNIT TESTS

A good first step would be to copy the generated unit tests into the project test source directory. The unit tests are generated by `camel-api-component-maven-plugin` when you invoke `mvn install`

and if there isn't already a test case in the `src/test/java/packagename` directory. In this case, you have a test case for the Javadoc- and file-supplemented APIs in the `target/generated-test-sources/camel-component/camelinaction` directory:

- `HiWorldFileHelloIntegrationTest.java`
- `HiWorldJavadocHelloIntegrationTest.java`

Both of these need to be copied into `src/test/java/camelinaction` so your edits won't be lost the next time the plugin is run.

These test cases don't do much as is. They mainly set up sample routes and show how parameter data is passed into the various APIs. Your job is to fill in the data with something useful.

For the file-supplemented API, your routes look like this:

```
@Override
protected RouteBuilder createRouteBuilder() throws
Exception {
    return new RouteBuilder() {
        public void configure() {
            // test route for greetMe
            from("direct://GREETME")
                .to("hiworld://" + PATH_PREFIX + "/greetMe?
inBody=name");

            // test route for greetUs
            from("direct://GREETUS")
                .to("hiworld://" + PATH_PREFIX +
"/greetUs");

            // test route for sayHi
            from("direct://SAYHI")
                .to("hiworld://" + PATH_PREFIX + "/sayHi");
        }
    };
}
```

Notice that there isn't a route with a `hi-world` consumer—no `from("hiworld...")`—you'll have to fill that in yourself if the component will support it. A test case is created for each API

method. For the `greetUs` method, the test case looks like this:

```
@Ignore
@Test
public void testGreetUs() throws Exception {
    final Map<String, Object> headers = new HashMap<String, Object>();
    // parameter type is String
    headers.put("CamelHiWorld.name1", null);
    // parameter type is String
    headers.put("CamelHiWorld.name2", null);

    final String result
        = requestBodyAndHeaders("direct://GREETUS", null,
headers);

    assertNotNull("greetUs result", result);
    LOG.debug("greetUs: " + result);
}
```

There isn't much to this test case, but it's handy because it shows you how many parameters to set as well as what their names are. When you've filled in some data and additional asserts, you can remove the `@Ignore` annotation.

FILLING IN THE OTHER BLANKS

Now that you've seen how the tests are set up, let's look at the component. Recall that the sample `hi-world-component` contains several source files under `src/main/java/camelinaction`:

```
hi-world-component
└── pom.xml
└── src
    └── main
        └── java
            └── camelinaction
                ├── HiWorldComponent.java
                ├── HiWorldConfiguration.java
                ├── HiWorldConsumer.java
                ├── HiWorldEndpoint.java
                └── HiWorldProducer.java
```

Many of these classes have methods you can override or tweak for your own component. Most are optional, however. A good

place to start is `HiworldEndpoint`, and in particular the `afterConfigureProperties` method:

```
@Override  
protected void afterConfigureProperties() {  
    switch (apiName) {  
        case HELLO_FILE:  
            apiProxy = new HiWorldFileHello();  
            break;  
        case HELLO_JAVADOC:  
            apiProxy = new HiWorldJavadocHello();  
            break;  
        default:  
            throw new IllegalArgumentException("Invalid API  
name " + apiName);  
    }  
}
```

Here you set `apiProxy` to the proxy object for the particular API being used in the current endpoint. This switch block matches up with the configuration for the `camel-api-component-maven-plugin`:

```
<api>  
    <apiName>hello-file</apiName>  
  
    <proxyClass>camelinaction.api.HiWorldFileHello</proxyClass>  
        <fromSignatureFile>signatures/file-sig-  
api.txt</fromSignatureFile>  
    </api>  
    <api>  
        <apiName>hello-javadoc</apiName>  
  
        <proxyClass>camelinaction.api.HiWorldJavadocHello</proxyCla  
ss>  
        <fromJavadoc/>  
    </api>
```

If this matches up, what's left to do? In many cases, the Java library containing this proxy class needs to be configured. This could include things like authentication for a remote server, opening a session, or setting connection properties. These things often differ for each third-party library, so the API component framework can't figure this out for you.

If you still feel totally lost with how to proceed after the framework is done with its generation work, you may find inspiration from existing components supported by the API component framework. These are as follows:

- camel-box
- camel-braintree
- camel-google-calendar
- camel-google-drive
- camel-google-pubsub
- camel-google-mail
- camel-linkedin
- camel-olingo2
- camel-olingo4
- camel-twilio
- camel-zendesk

By now you should have a good understanding of how to write new components for Apache Camel. Hopefully, you'll also consider contributing them back to the project! (Appendix B covers contributing to Apache Camel.) Let's now consider another extension point in Camel: data formats.

8.6 Developing data formats

As you saw in chapter 3, *data formats* are pluggable transformers that can transform messages from one form to another, and vice versa. Each data format is represented in Camel as an interface in `org.apache.camel.spi.DataFormat` containing two methods:

- `marshal`—For marshaling a message into another form, such as marshaling Java objects to XML, CSV, EDI, HL7, JSON or other

well-known data models

- `unmarshal`—For performing the reverse operation, which turns data from well-known formats back into a message

Camel has many data formats, but at times you might need to write your own. In this section, you'll look at how to develop a data format that can reverse strings. Let's first set up the initial boilerplate code using an archetype.

8.6.1 GENERATING THE SKELETON DATA FORMAT PROJECT

To generate a skeleton project by using the Camel API component framework, you need to use the `camel-archetype-dataformat` archetype. Try this with following Maven command:

```
mvn archetype:generate \
  -B \
  -DarchetypeGroupId=org.apache.camel.archetypes \
  -DarchetypeArtifactId=camel-archetype-dataformat \
  -DarchetypeVersion=2.20.1 \
  -DgroupId=camelinaaction \
  -DartifactId=reverse-dataformat \
  -Dname=Reverse \
  -Dscheme=reverse
```

The resulting `reverse-dataformat` directory layout is shown in the following listing.

Listing 8.9 Layout of the project created by `camel-archetype-api-component`

```
reverse-dataformat
├── pom.xml
└── ReadMe.txt
└── src
    └── main
        └── java
            └── camelinaction
                └── ReverseDataFormat.java
```

1

1

Skeleton data format class

```
└ resources
  └ META-INF
    └ services
      └ org
        └ apache
          └ camel
            └ dataformat
              └ reverse
```

②

②

Ensure custom data format is registered with “reverse” name in Camel

```
└ test
  └ java
    └ camelaction
      └ ReverseDataFormatTest.java
```

③

③

Test case for data format

```
└ resources
  └ log4j.properties
```

This data format is usable right away and even has a working test case. But it doesn’t do much—it just returns the data unmodified. Let’s see how to make your data format change the data.

8.6.2 WRITING THE CUSTOM DATA FORMAT

Developing your own data format is fairly easy, because Camel provides a single API you must implement:

`org.apache.camel.spi.DataFormat`. Let’s look at how to implement a string-reversing data format, shown in the following listing.

Listing 8.10 Developing a custom data format that can reverse strings

```
public class ReverseDataFormat
```

```
extends ServiceSupport
implements DataFormat, DataFormatName {

    public String getDataFormatName() {
        return "reverse";
    }

    public void marshal(Exchange exchange, Object graph,
        OutputStream stream) throws Exception { 1
}
```

1

Marshals to reverse string

```
    byte[] bytes
    =
exchange.getContext().getTypeConverter().mandatoryConvertTo
(
    byte[].class, graph);
String body = reverseBytes(bytes);
stream.write(body.getBytes());
}

public Object unmarshal(Exchange exchange, InputStream
stream)
throws Exception { 2
```

2

Unmarshals to unreverse string

```
    byte[] bytes =
exchange.getContext().getTypeConverter().
mandatoryConvertTo(byte[].class, stream);
String body = reverseBytes(bytes);
return body;
}

private String reverseBytes(byte[] data) {
    StringBuilder sb = new StringBuilder(data.length);
    for (int i = data.length - 1; i >= 0; i--) {
        char ch = (char) data[i];
        sb.append(ch);
    }
    return sb.toString();
}
```

```

@Override
protected void doStart() throws Exception {
    // init logic here
}

@Override
protected void doStop() throws Exception {
    // cleanup logic here
}

}

```

The custom data format must implement the `DataFormat` interface, which forces you to develop two methods: `marshal` and `unmarshal`. That's no surprise, because they're the same methods you use in the route. The `marshal` method ① needs to output the result to `OutputStream`. To do that, you need to get the message payload as a `byte[]` and then reverse it with a helper method. Then you write that data to `OutputStream`. Note that you use the Camel type converters to return the message payload as a `byte[]`. This is powerful and saves you from doing a manual typecast in Java or trying to convert the payload yourself.

The `unmarshal` method ② is nearly the same. You use the Camel type-converter mechanism again to provide the message payload as a `byte[]`. `unmarshal` also reverses the bytes to get the data back in its original order. Note that in this method you return the data instead of writing it to a stream.

TIP As a best practice, use the Camel type converters instead of typecasting or converting between types yourself. Chapter 3 covers Camel's type converters.

To use this new data format in a route, all you have to do is define it as a bean and refer to it by using `<custom>` as follows:

```

<bean id="reverse"
class="camelinaction.ReverseDataFormat"/>

<camelContext id="camel"

```

```

xmlns="http://camel.apache.org/schema/spring">
<route>
    <from uri="direct:marshal"/>
    <marshal>
        <custom ref="reverse"/>
    </marshal>
    <to uri="log:marshal"/>
</route>

<route>
    <from uri="direct:unmarshal"/>
    <unmarshal>
        <custom ref="reverse"/>
    </unmarshal>
    <to uri="log:unmarshal"/>
</route>
</camelContext>

```

Using the Java DSL looks like this:

```

from("direct:marshal")
    .marshal().custom("reverse")
    .to("log:marshal");
from("direct:unmarshal")
    .unmarshal().custom("reverse")
    .to("log:unmarshal");

```

Alternatively, you can pass in an instance of the data format directly:

```

DataFormat format = new ReverseDataFormat();
from("direct:in").marshal(format);
from("direct:back").unmarshal(format).to("mock:reverse");

```

You'll find this example in the chapter8/reverse-dataformat directory, and you can try it by using the following Maven goal:

```
mvn test
```

At this point, you should have a good idea of what it takes to create a new Camel data format.

8.7 Summary and best practices

Knowing how to create Camel projects is important, and you

may have wondered why we chose to discuss this so late in the book. We felt it was best to focus on the core concepts first and worry about project setup details later. Also, you should now have a better idea of the cool Camel applications you can create, having read about the features first. At this point, you should be well equipped to start your own Camel application and make it do useful work.

Before we move on, here are the key ideas to take away from this chapter:

- *The easiest way to create Camel applications is with Maven archetypes*—Nothing is worse than having to type out a bunch of boilerplate code for new projects. The Maven archetypes provided by Camel and the Spring Boot start website (<http://start.spring.io>) will get you started much faster.
- *The easiest way to manage Camel library dependencies is with Maven archetypes*—Camel is just a Java framework, so you can use whatever build system you like to develop your Camel projects. Using Maven will eliminate many of the hassles of tracking down JAR files from remote repos, letting you focus more on your business code than the library dependencies.
- *IDEs like Eclipse or IDEA make Camel development easier*—From autocompletion of DSL methods to great Maven integration, developing Camel projects is a lot easier within an IDE.
- *Camel has features that assist you when debugging*—You don't always have to dive deep into the Camel source in an IDE to solve bugs. Camel provides a great tracing facility, many JMX operations and attributes, and logging. If you're in a test case, Camel also provides a debugger facility that allows you to step through each node in the runtime processor graph. You can even take the easy road and use a tooling project such as hawtio or JBoss Fuse Tooling to visually debug your routes.
- *If you find no component in Camel for your use case, create your own*—Camel allows you to write and load up your own

custom components easily. There's even a Maven archetype for starting a custom component project.

- *Consider using the API component framework when creating new components*—If you need to create a new component using an existing third-party library, the API component framework can generate much of the component for you.
- *If you find no data format in Camel for your use case, create your own*—As for components, Camel allows you to write and load your own custom data formats easily. There's also a Maven archetype for starting a custom data-format project.

In the next chapter, we'll look at a topic that can help make you a successful integration specialist: testing with Camel. Without it, you'll almost certainly be in trouble. We'll also look at simulating errors to test whether your error-handling strategies work as expected.

9

Testing

This chapter covers

- Introducing and using the Camel Test Kit
- Unit-testing Camel
- Testing with mocks
- Simulating components and errors
- Amending routes before testing
- Integration and system testing
- Using third-party test frameworks with Camel

Integrated systems are notoriously difficult to test. Systems being built and integrated today often involve a myriad of systems ranging from legacy systems, closed source commercial products, homegrown applications, and so on. These systems often have no built-in support for automated testing.

The poor folks who are tasked with testing such integrated systems often have no other choice than to play through manual use cases—triggering events from one window/terminal and then crossing their fingers and watching the results in the affected systems. And by *watching*, we mean looking at log files, checking databases, or any other manual procedure to verify the results. When they move on to the next use case, they have to *reset the*

system to ensure its correct state before carrying on with the testing.

During development, the systems being integrated are often test systems that are from other teams or are available only in production. If the systems aren't network connected, internal company procedures may hold back a speedy resolution. With the higher number of systems involved, it becomes more difficult for developers to build and test their work.

Does this sound familiar? It should, because these problems have been around for decades, and people have found ways around them in order to do their jobs. One such way is to stub out other systems and not rely on their physical implementations. Camel has answers for this with a built-in test kit that allows you to treat integration points as components that can be switched out with local testable implementations.

This chapter starts by introducing the Camel Test Kit and teaching you to perform Camel testing with various runtime platforms such as standalone Java, Spring Boot, CDI, OSGi, and Java EE servers such as WildFly.

At this point in this book, you've likely looked at the source code examples and seen that the vast majority of the code is driven by unit tests. Therefore, you may have seen that some of these tests are using the Camel mock component. We'll dive deeper into what the mock component is, what functionality it offers for testing, and how to get more out of it in your tests.

We've said it before: integration is hard. Building and integrating systems would be easier if nothing ever went wrong. But what about when things do go wrong? How do you test your systems when a remote system is unavailable, when invalid data is returned, or when a database starts failing with SQL violation errors? This chapter covers how to simulate such error conditions in your tests.

Integration systems start to add value to your business only when they become live in production. You may wonder how you

can build Camel routes that run in production but are testable both locally and in test environments. In this chapter, you'll learn the strategies that the Camel Test Kit offers for testing Camel routes that are built as production-ready routes.

At the end of the chapter, we'll look at three third-party testing frameworks for doing integration testing and learn how to use them with your Camel applications.

Other testing disciplines that we don't cover in this book are worth keeping in mind, such as load, performance, and stress testing. Recently, companies have started seeing the benefits of having a continuous integration and deployment platform that helps drive test automation and makes organizations more agile while reducing time to production.

Testing starts with unit tests. And a good way to perform unit testing on a Camel application is to start the application, send messages to the application, and verify that the messages are routed as expected. This is illustrated in [figure 9.1](#). You send a message to the application, which transforms the message to another format and returns the output. You can then verify that the output is as expected.

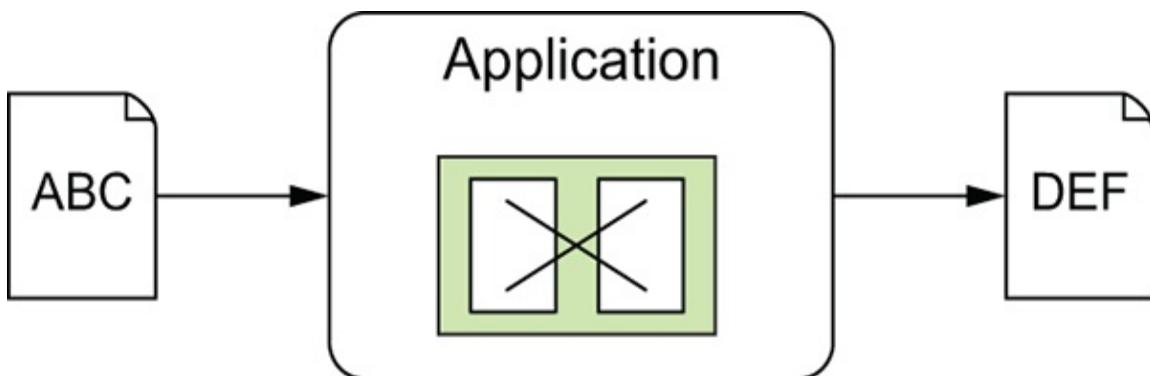


Figure 9.1 Testing a Camel application by sending a message to the application and then verifying the returned output

This is how the Camel Test Kit is used for testing. You'll learn to set up expectations as preconditions for your unit tests, start the tests by sending in messages, and verify the results to determine whether the tests passed.

9.1 Introducing the Camel Test Kit

Camel provides rich facilities for testing your projects, using the Camel Test Kit. This kit is used heavily for testing Camel itself (Camel is eating its own dog food). [Figure 9.2](#) gives a high-level overview of the Camel Test Kit.

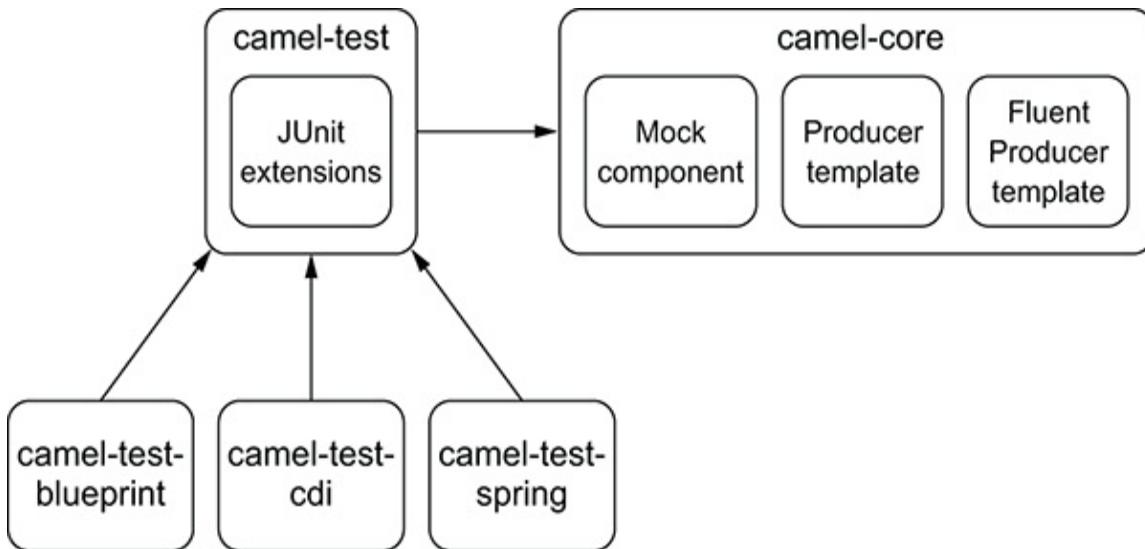


Figure 9.2 The Camel Test Kit is comprised of the camel-test JAR for testing with Java in general, and three specialized JARs for testing with OSGi Blueprint, CDI, and Spring. Inside camel-core you'll find the mock component, ProducerTemplate, and FluentProducerTemplate, which are frequently used with unit testing.

The test kit has four main parts. The camel-test JAR is the core test module that includes the JUnit extensions as a number of classes on top of JUnit that make unit testing with Camel much easier. We cover them in the next section. The camel-test JAR can be used for testing standalone Camel applications. Testing Camel in other environments requires specialized camel-test modules. For OSGi Blueprint testing, you should use camel-test-blueprint, for CDI testing use camel-test-cdi, and for Camel with Spring use camel-test-spring. This section focuses on testing standalone Camel applications using plain Java. Section 9.2 covers testing Camel with Spring, OSGi Blueprint, and CDI.

The Camel Test Kit uses the mock component from camel-core, covered in section 9.3. And you're already familiar with

`ProducerTemplate` and `FluentProducerTemplate` that makes it easy to send messages to your Camel routes.

Let's now look at the Camel JUnit extensions and see how to use them to write Camel unit tests.

9.1.1 USING THE CAMEL JUNIT EXTENSIONS

What are the Camel JUnit extensions? They are nine classes in a small JAR file, `camel-test.jar`, that ships with Camel. Out of those nine classes are three that are intended to be used by end users. They are listed in table 9.1.

Table 9.1 End-user-related classes in the Camel Test Kit, provided in `camel-test.jar`

Class	Description
<code>org.apache.camel.test.junit4.TestSupport</code>	Abstract base class with additional builder and assertion methods.
<code>org.apache.camel.test.junit4.CamelTestSupport</code>	Base test class prepared for testing Camel routes. This is the test class you should use when testing your Camel routes.
<code>org.apache.camel.test.AvailablePortFinder</code>	Helper class to find unused TCP port numbers that your unit tests can use when testing with TCP connections.

To get started with the Camel Test Kit, you'll use the following route for testing:

```
from("file:inbox").to("file:outbox");
```

This is the “Hello World” example for integration kits that moves files from one folder to another. How do you go about unit testing this route?

You could do it the traditional way and write unit test code with the plain JUnit API. This would require at least 30 lines of code, because the API for file handling in Java is low level, and you need a fair amount of code when working with files.

An easier solution is to use the Camel Test Kit. In the next couple of sections, you'll work with the `CamelTestSupport` class—

the easiest to start with.

9.1.2 USING CAMEL-TEST TO TEST JAVA CAMEL ROUTES

In this chapter, we've kept the dependencies low when using the Camel Test Kit. All you need to include is the following dependency in the Maven pom.xml file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-test</artifactId>
    <version>2.20.1</version>
    <scope>test</scope>
</dependency>
```

Let's try it. You want to build a unit test to test a Camel route that copies files from one directory to another. The unit test is shown in the following listing.

Listing 9.1 A first unit test using the Camel Test Kit

```
public class FirstTest extends CamelTestSupport {

    @Override
    protected RouteBuilder createRouteBuilder() throws
Exception {
    return new RouteBuilder() {
        public void configure() throws Exception { ❶
```

❶

Defines route to test

```
        from("file://target/inbox")
            .to("file://target/outbox");
    }
};

@Test
public void testMoveFile() throws Exception {
    template.sendBodyAndHeader("file://target/inbox",
"Hello World",
```

```
Exchange.FILE_NAME, "hello.txt");
```

②

②

Creates hello.txt file

```
Thread.sleep(2000);  
  
File target = new File("target/outbox/hello.txt");  
assertTrue("File not moved", target.exists());
```

③

③

Verifies file is moved

```
}
```

The `FirstTest` class must extend the `org.apache.camel.junit4.CamelTestSupport` class to conveniently use the Camel Test Kit. By overriding the `createRouteBuilder` method, you can provide any route builder you wish. You use an inlined route builder, which allows you to write the route directly within the unit-test class. All you need to do is override the `configure` method ① and include your route.

The test methods are regular JUnit methods, so the method must be annotated with `@Test` to be included when testing. You'll notice that the code in this method is fairly short. Instead of using the low-level Java File API, this example uses Camel as a client by using `ProducerTemplate` to send a message to a file endpoint ②, which writes the message as a file.

In the test, you sleep 2 seconds after dropping the file in the inbox folder; this gives Camel a bit of time to react and route the file. By default, Camel scans twice per second for incoming files, so you wait 2 seconds to be on the safe side. Finally, you assert that the file was moved to the outbox folder ③.

The book's source code includes this example. You can try it on your own by running the following Maven goal from the

chapter9/java directory:

```
mvn test -Dtest=FirstTest
```

When you run this example, it should output the result of the test as shown here:

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

This indicates that the test completed successfully; there are no failures or errors.

IMPROVING THE UNIT TEST

The unit test in [listing 9.1](#) could be improved in a few areas, such as ensuring that the starting directory is empty and that the written file's content is what you expect.

The former is easy, because the `CamelTestSupport` class has a method to delete a directory. You can do this in the `setUp` method:

```
public void setUp() throws Exception {  
    deleteDirectory("target/inbox");  
    deleteDirectory("target/outbox");  
    super.setUp();  
}
```

Camel can also test the written file's content to ensure that it's what you expect. You may remember that Camel provides an elaborate type-converter system, and that this system goes beyond converting between simple types and literals. The Camel type system includes file-based converters, so there's no need to fiddle with the various cumbersome Java I/O file streams. All you need to do is ask the type-converter system to grab the file and return it to you as a `String`.

Just as you had access to the template in [listing 9.1](#), the Camel Test Kit also gives you direct access to `CamelContext`. The `testMoveFile` method in [listing 9.1](#) could have been written as follows:

```
@Test
public void testMoveFile() throws Exception {
    template.sendBodyAndHeader("file://target/inbox", "Hello
World",
                                Exchange.FILE_NAME, "hello.txt");

    Thread.sleep(2000);

    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());
```

Asserts the file is moved

```
String content = context.getTypeConverter()
                    .convertTo(String.class, target);
assertEquals("Hello World", content);
```

Asserts the file content is Hello World

}

Is there one thing in the test code that annoys you? Yeah, `Thread.sleep(2000)`, to wait for Camel to pick up and process the file. Camel can do better. You can use something called `NotifyBuilder` to set up a precondition using the builder and then let it wait until the condition is satisfied before continuing. We cover `NotifyBuilder` in section 9.5.2, so we'll just quickly show you the revised code:

```
@Test  
public void testMoveFile() throws Exception {  
    NotifyBuilder notify = new NotifyBuilder(context)  
        .whenDone(1).create();
```

Sets up precondition on NotifyBuilder

```
assertTrue(notify.matchesMockWaitTime());
```

NotifyBuilder is waiting for the precondition to be satisfied.

```
File target = new File("target/outbox/hello.txt");
assertTrue("File not moved", target.exists());
```

Asserts the file is moved

```
String content = context.getTypeConverter()
    .convertTo(String.class, target);
assertEquals("Hello World", content);
```

Asserts the file content is Hello World

```
}
```

The book's source code includes this revised example. You can try it by running the following Maven goal from the chapter9/java directory:

```
mvn test -Dtest=FirstNoSleepTest
```

The preceding examples cover the case in which the route is defined in the unit-test class as an anonymous inner class. But what if you have a route defined in another class? How do you go about unit testing that route instead? Let's look at that next.

9.1.3 UNIT TESTING AN EXISTING ROUTEBUILDER CLASS

Defining Camel routes in separate RouteBuilder classes is common, as in the FileMoveRoute class here:

```
public class FileMoveRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("file://target/inbox")
            .to("file://target/outbox");
```

```
    }  
}
```

How could you unit test this route from the `FileMoveRoute` class? You don't want to copy the code from the `configure` method into a JUnit class. Fortunately, it's easy to set up unit tests that use the `FileMoveRoute`, as you can see here:

```
protected RouteBuilder createRouteBuilder() throws  
Exception {  
    return new FileMoveRoute();  
}
```

Yes, it's that simple! Just return a new instance of your route class.

This book's source code contains this example in the `chapter9/java` directory. You can try the example by using the following Maven goal:

```
mvn test -Dtest=ExistingRouteBuilderTest
```

Notice that the `FileMoveRoute` class is located in the `src/main/java/camelinaction` directory and isn't located in the test directory.

Now you've learned how to use `CamelTestSupport` for unit testing routes based on the Java DSL. You use `CamelTestSupport` from `camel-test` for testing standard Java-based Camel applications. But Camel can run in many platforms. Let's dive into covering the most popular choices.

9.2 Testing Camel with Spring, OSGi, and CDI

Camel applications are being used in many environments and together with various frameworks. Over the years, people have often run Camel with either Spring or OSGi Blueprint, and more recently in using CDI with Java EE applications. This section demonstrates how to start building unit tests with the following frameworks and platforms:

- Camel using Spring XML
- Camel using Spring Java Config
- Camel using Spring Boot
- Camel using OSGi Blueprint XML running in Apache Karaf/ServiceMix, or JBoss Fuse
- Camel using CDI running standalone using JBoss Weld CDI container
- Camel using CDI running in WildFly Swarm
- Camel using CDI running in WildFly

It's worth emphasizing that the goal of this section is to show you how to get a good start in testing Camel on these seven platforms. The focus is therefore on using a simple test case as an example to highlight how the testing is specific to the chosen platform. We'll dive deeper into testing topics such as integration and systems tests, and more elaborate test functionality from Camel, in the sections to follow.

9.2.1 CAMEL TESTING WITH SPRING XML

You use camel-test-spring to test Spring-based Camel applications. In the Maven pom.xml file, add the following dependency:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test-spring</artifactId>
  <version>2.20.1</version>
  <scope>test</scope>
</dependency>
```

This section looks at unit-testing the route in the following listing.

[Listing 9.2](#) A Spring-based version of the route in [listing 9.1](#) (firststep.xml)

```

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

    <camelContext id="camel"
      xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="file:///target/inbox"/>
            <to uri="file:///target/outbox"/>
        </route>
    </camelContext>

</beans>

```

How can you unit-test this route? Ideally, you should be able to use unit tests regardless of the language used to define the route —whether using Java DSL or XML DSL, for example.

The camel-test-spring module is an extension to camel-test, which means all the test principles you learned in section 9.1 also apply. Camel is able to handle this; the difference between using `SpringCamelTestSupport` and `CamelTestSupport` is just a matter of how the route is loaded. The unit test in the following listing illustrates this point.

Listing 9.3 A first unit test using Spring XML routes

```

public class SpringFirstTest extends CamelSpringTestSupport
{
    protected AbstractXmlApplicationContext
createApplicationContext() {
    return new ClassPathXmlApplicationContext(
    "camelinaction/firststep.xml");
}

```

1

Loads Spring XML file

```
}
```

```
@Test
public void testMoveFile() throws Exception {
    template.sendBodyAndHeader("file://target/inbox",
        "Hello World", Exchange.FILE_NAME,
    "hello.txt");

    Thread.sleep(2000);

    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());
    String content = context.getTypeConverter()
        .convertTo(String.class,
    target);
    assertEquals("Hello World", content);
}
}
```

You extend the `CamelSpringTestSupport` class so you can unit-test with Spring XML-based routes. You need to use a Spring-based mechanism to load the routes ①; you use the `ClassPathXmlApplicationContext`, which loads your route from the classpath. This mechanism is entirely Spring based, so you can also use the `FileSystemXmlApplicationContext`, include multiple XML files, and so on; Camel doesn't impose any restrictions. The `testMoveFile` method is exactly the same as it was when testing using Java DSL in section 9.1, which means you can use the same unit-testing code regardless of how the route is defined.

This book's source code contains this example in the `chapter9/spring-xml` directory; you can try the example by using the following Maven goal:

```
mvn test -Dtest=SpringFirstTest
```

UNIT TESTING WITH @RUNWITH

The test class in [listing 9.3](#) extends the `CamelSpringTestSupport` class. But you can also write Camel unit tests without extending a base class, which is a more modern style with JUnit. The following listing shows how to write the unit test in a more modern style.

[Listing 9.4](#) Using a modern JUnit style to write a Camel Spring unit test

```
@RunWith(CamelSpringRunner.class) 1
```

1

Runs the test using Camel Spring runner

```
@ContextConfiguration(locations = {"firststep.xml"}) 2
```

2

Declares the location of the Spring XML file relative to this unit test

```
public class SpringFirstRunWithTest {
```

```
    @Autowired
```

```
    private CamelContext context; 3
```

3

Dependency injects CamelContext

```
    @Autowired
```

```
    private ProducerTemplate template; 4
```

4

Dependency injects ProducerTemplate

```
    @Test
```

```
    public void testMoveFile() throws Exception { 5
```

5

Test method same as in listing 9.3

```
template.sendBodyAndHeader("file://target/inbox",
                           "Hello World", Exchange.FILE_NAME,
                           "hello.txt");

Thread.sleep(2000);

File target = new File("target/outbox/hello.txt");
assertTrue("File not moved", target.exists());
String content = context.getTypeConverter()
                           .convertTo(String.class,
                           target);
assertEquals("Hello World", content);
}
}
```

Instead of extending a base class, the test class is annotated with `@RunWith(CamelSpringRunner.class)` ❶. The `CamelSpringRunner` is an extension to Spring's `SpringJUnit4ClassRunner` that integrates Camel with Spring and allows you to use additional Camel annotations in the test. We cover such use cases later in this chapter.

The `@ContextConfiguration` annotation is used to configure where the Spring XML file is located in the classpath ❷. Notice that the location is relative to the current class, so the `firststep.xml` file is also located in the `camelinaction` package.

To use Camel during the unit test, you need to dependency inject both `camelContext` ❸ and `ProducerTemplate` ❹. The test method is unchanged ❺.

This example is also provided with the source code in the `chapter9/spring-xml` directory, and you can try the example by running the following Maven goal:

```
mvn test -Dtest=SpringFirstRunWithTest
```

Now let's move on to testing Spring applications that are using Java Config instead of XML.

9.2.2 CAMEL TESTING WITH SPRING JAVA

CONFIG

The Spring Framework started out mainly using XML configuration files, which became a popular choice of building applications. Camel works well with XML by offering the XML DSL so you can define Camel routes directly in your Spring XML files. But recently, an alternative configuration using plain Java has become popular as well—with a lot of thanks to the rising popularity of Spring Boot.

When using Camel with Spring Java Config, you must remember to add the following dependency to your Maven pom.xml file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-javaconfig</artifactId>
    <version>2.20.1</version>
</dependency>
```

You can use Spring Java Config both standalone and with Spring. We cover both scenarios in this and the following section.

The example we use is `FileMoveRoute`, as shown here:

`@Component`

①

①

Annotates the class with `@Component` to make Spring discover the class easily

```
public class FileMoveRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("file://target/inbox")
            .to("file://target/outbox");
    }
}
```

The `FileMoveRoute` class is a plain Camel route that has been annotated with `@Component` ①. This enables Spring Java Config to discover the route during startup and automatically create an

instance that in turn Camel discovers from Spring and automatically includes the route.

To bootstrap and run Camel with Spring Java Config, you need the following:

- A Java main class to start the application
- An `@Configuration` class to bootstrap Spring Java Config
- You can combine both in the same class, as shown in the following listing.

Listing 9.5 Java main class to bootstrap Spring Java Config with Camel

```
@Configuration      ①
```

1

Marks this class as a Spring Java configuration class

```
@ComponentScan("camelinaction")      ②
```

2

Tells Spring to scan in the package for classes with Spring annotations

```
public class MyApplication extends CamelConfiguration  
{      ③
```

3

To automatically enable Camel

```
public static void main(String[] args) throws Exception {  
    Main main = new Main();      ④
```

4

Main class to easily start the application and keep the JVM running

```
main.setConfigClass(MyApplication.class);
```

5

5

Uses this class as the configuration class

```
} main.run();  
}
```

The `MyApplication` class extends `CamelConfiguration` ③ and has been marked as a `@Configuration` ① class. This allows you to configure the application in this class by using Spring Java Config. But this example is simple, and there's no additional configuration.

TIP You can add methods annotated with `@Bean` in the configuration class to have Spring automatically call the methods and enlist the returned bean instances into the Spring bean context.

To include the Camel routes, you enable `@ComponentScan` ② to scan for classes that have been annotated with `@Component`; these would automatically be enlisted into the Spring bean context. This in turn allows Camel to discover the routes from Spring and automatically include them when Camel starts up.

To make it easy to run this application standalone, you use a `Main` class from `camel-spring-javaproject` that in a few lines of code start the application and keep the JVM running ④ ⑤.

How do you unit-test this application? You can test the application using the same `CamelSpringTestSupport` you used when testing Camel with Spring XML files. The difference is in how you create Spring in the `createApplicationContext` method, as shown in the following listing.

Listing 9.6 Testing a Spring Java Config application using `CamelSpringTestSupport`

```
public class FirstTest extends CamelSpringTestSupport {  
    @Override  
    protected AbstractApplicationContext  
createApplicationContext() {  
    AnnotationConfigApplicationContext acc = new  
AnnotationConfigApplicationContext(); ❶  
}
```

❶

Creates a Spring Java Config context

```
acc.register(MyApplication.class); ❷
```

❷

Registers the @Configuration class in the Spring context

```
    return acc;  
}
```

```
@Test  
public void testMoveFile() throws Exception { ❸
```

❸

The unit test method is identical with testing using Spring XML files

```
template.sendBodyAndHeader("file://target/inbox",  
                           "Hello World", Exchange.FILE_NAME,  
                           "hello.txt");  
  
Thread.sleep(2000);  
  
File target = new File("target/outbox/hello.txt");  
assertTrue("File not moved", target.exists());  
String content = context.getTypeConverter()  
                           .convertTo(String.class,  
target);  
assertEquals("Hello World", content);  
}  
}
```

The main difference is that Spring Java Config uses

`AnnotationConfigApplicationContext` ❶ as the Spring bean container. You then have to configure which `@Configuration` class to use using the `register` method ❷. The rest of the unit test is identical ❸ to [listing 9.3](#), which was testing using Spring XML.

The book's source code contains this example in the `chapter9/spring-java-config` directory; you can try the example by using the following:

```
mvn test -Dtest=FirstTest
```

You can also unit-test without extending the `CamelSpringTestSupport` class, but instead using a set of annotations. This started becoming more popular recently when JUnit 4.x introduced this style.

UNIT TESTING WITH `@RunWith`

Instead of extending the `CamelSpringTestSupport` class, you can use the `CamelSpringRunner` as a JUnit runner, which will run the unit tests with Camel support. The following listing shows how to do that.

Listing 9.7 Unit testing with `@RunWith` instead of extending the `CamelSpringTestSupport` class

```
@RunWith(CamelSpringRunner.class) ❶
```

❶

Uses `@RunWith` with the `CamelSpring` runner

```
@ContextConfiguration( ❷
```

❷

Configures Spring to use the `@Configuration` class

```
classes = MyApplication.class, ❸
```

❸

```
        loader =  
CamelSpringDelegatingTestContextLoader.class) ②  
public class RuntWithFirstTest {  
  
    @Autowired  
    private CamelContext context; ③
```

③

Dependency injects CamelContext and ProducerTemplate

```
@Autowired ③  
private ProducerTemplate template; ③  
  
@Test  
public void testMoveFile() throws Exception { ④
```

④

The unit test method is identical to testing without using @RunWith.

```
template.sendBodyAndHeader("file://target/inbox",  
                           "Hello World", Exchange.FILE_NAME,  
                           "hello.txt");  
  
Thread.sleep(2000);  
  
File target = new File("target/outbox/hello.txt");  
assertTrue("File not moved", target.exists());  
String content = context.getTypeConverter()  
                           .convertTo(String.class,  
target);  
assertEquals("Hello World", content);  
}  
}
```

The unit test uses CamelSpringRunner as the @RunWith runner ①. To configure Spring, you use @ContextConfiguration ②; here you refer to the class that holds @Configuration. Then the loader refers to CamelSpringDelegatingTestContextLoader, which is a class that enables using a set of additional annotations to enable certain features during testing. In this simple example, you're not using any of these features and can omit declaring the loader. But including the loader is a good habit, so you're ready

for using Camel-testing capabilities such as automocking and advice. These capabilities are revealed later in this chapter.

Then you dependency-inject `CamelContext` and `ProducerTemplate` ❸, which you use in the unit-test method ❹. You have to do this because the test class no longer extends `CamelTestSpringSupport`, which provides `CamelContext` and `ProducerTemplate` automatically.

You can try this example in the book's source code, in the `chapter9/spring-java-config` directory, by using the following Maven goal:

```
mvn test -Dtest=RunWithFirstTest
```

What you've learned here about using Spring Java Config is also relevant when using Spring Boot. All Spring Boot applications use the Java configuration style, so let's see how you can build Camel unit tests with Bootify Spring—eh, Spring Boot.

9.2.3 CAMEL TESTING WITH SPRING BOOT

Spring Boot makes Spring Java Config applications easier to use, develop, run, and test. Using Camel with Spring Boot is easy: you add the following dependency to your Maven `pom.xml` file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-boot-starter</artifactId>
  <version>2.20.1</version>
</dependency>
```

And as when using Spring Java Config, your Camel routes can be automatically discovered if you annotate the class with `@Component`:

```
@Component ❶
```

❶

Annotate the class with `@Component` to make Spring discover the class easily.

```
public class FileMoveRoute extends RouteBuilder {  
    @Override  
    public void configure() throws Exception {  
        from("file://target/inbox")  
            .to("file://target/outbox");  
    }  
}
```

Running Camel on Spring Boot is easy. Spring Boot will automatically embed Camel and all its Camel routes (annotated with `@Component`). To set up a Spring Boot application, all you need to do is type the following lines of Java code:

```
@SpringBootApplication ①
```

①

SpringBootApplication annotation to mark this class as a Spring Boot application

```
public class MyApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(MyApplication.class,  
args); ②
```

②

One line to start the Spring Boot application

```
}
```

All you have to do is annotate the class with `@SpringBootApplication` ① and write one line of Java code in the `main` method ② to run the application.

Testing this application is as easy as testing the Spring Java Config application covered in the previous section. The only difference is that with Spring Boot, you use `@SpringBootTest` instead of `@ContextConfiguration` to specify which configuration class to use. The following listing shows the Camel unit test with Spring Boot.

Listing 9.8 Unit testing Camel with Spring Boot

```
@RunWith(CamelSpringBootRunner.class) 1
```

1

Uses `@RunWith` with the `CamelSpringBoot` runner

```
@SpringBootTest(classes = MyApplication.class) 2
```

2

Configures Spring Boot to use `MyApplication` as the configuration class

```
public class RunWithFirstTest {
```

```
    @Autowired
```

```
    private CamelContext context; 3
```

3

Dependency injects `CamelContext` and `ProducerTemplate`

```
    @Autowired 3
```

```
    private ProducerTemplate template; 3
```

```
    @Test
```

```
    public void testMoveFile() throws Exception { 4
```

4

The unit test method is identical to testing without using `@RunWith`.

```
        template.sendBodyAndHeader("file://target/inbox",
                                  "Hello World", Exchange.FILE_NAME,
                                  "hello.txt");

        Thread.sleep(2000);

        File target = new File("target/outbox/hello.txt");
        assertTrue("File not moved", target.exists());
        String content = context.getTypeConverter()
                            .convertTo(String.class,
                                      target);
```

```
        assertEquals("Hello World", content);
    }
}
```

To run the test as JUnit, you use `CamelSpringBootRunner` as `@RunWith` (notice that the runner is named `CamelSpringBootRunner`). Then you refer to the Spring Boot application class, which is `MyApplication` ②. The rest is similar to testing with Spring Java Config ④, where you must dependency inject `CamelContext` and `ProducerTemplate` ③ to make them available for use in your test methods.

We prepared this example in the source code in the `chapter9/spring-boot` directory, and you can try the example by using the following Maven goal:

```
mvn test -Dtest=FirstTest
```

TIP If you want to learn more about Spring Boot, see *Spring Boot in Action* by Craig Walls (Manning, 2016).

This example is a basic example with just one route in one `RouteBuilder` class. What if you have multiple `RouteBuilder` classes? How do you test them?

TESTING WITH ONLY ONE ROUTEBUILDER CLASS ENABLED

When you have multiple Camel routes in different `RouteBuilder` classes, Camel on Spring Boot will include all these routes. But you may want to write a unit test that's focused on testing the routes from only a specific `RouteBuilder` class.

Suppose you have two `RouteBuilder` classes named `FooRoute` and `BarRoute`, with related unit-test classes named `FooTest` and `BarTest`. How can you make `FooTest` include only `FooRoute`, and make `BarTest` only include `BarRoute`? Camel makes this easy, as you can use the following patterns to include (or exclude) which `RouteBuilders` to enable:

- `java-routes-include-pattern`—Used for including RouteBuilder classes that match the pattern.
- `java-routes-exclude-pattern`—Used for excluding RouteBuilder classes that match the pattern. Exclude takes precedence over include.

You can specify these patterns in your unit-test classes as properties to the `@SpringBootTest` annotation, as highlighted here:

```
@RunWith(CamelSpringBootRunner.class)
@SpringBootTest(classes = {MyApplication.class},
    properties = { "camel.springboot.java-routes-include-
pattern=**/Foo*"})
public class FooTest {
```

In the `FooTest` class, the include pattern is `**/Foo*`, which represents an *ant style* pattern. Notice that the pattern starts with double asterisk, which will match any leading package name. `/Foo*` means the class name must start with `Foo`, such as `FooRoute`, `FoolishRoute`, and so forth.

You can find this example in the `chapter9/spring-boot-test-one-route` directory, where you can run a test for each route by using the following Maven commands:

```
mvn test -Dtest=FooTest
mvn test -Dtest=BarTest
```

We've covered three ways of testing Camel with Spring. [Table 9.2](#) shows our recommendations for when to use what.

Table 9.2 Recommendations on using Camel with Spring XML, Spring Java Config, or Spring Boot

Spring	Recommendation
Spring XML	The Spring Framework has supported configuring applications using XML files for a long time. This is a well-established and used approach. This is a good choice if you prefer to use the XML DSL for defining your Camel routes.

Spring Java Config	Spring Java Configuration is less known and used even though it's been available for many years. Instead of using Spring Java Config, we recommend using Spring Boot if possible.
Spring Boot	Spring Boot is a popular choice these days and is a recommended approach for running Camel applications. You may think you must use Java code only with Spring Boot, but you can also use Spring XML files. This means you have freedom to define your Camel routes in both Java and XML.

Okay, that was a lot of coverage of testing Camel with Spring. Spring isn't all there is, so let's move on to testing with OSGi Blueprint.

9.2.4 CAMEL TESTING WITH OSGI BLUEPRINT XML

Camel supports defining Camel routes in OSGi Blueprint XML files. In this section, you'll learn how to start writing unit tests for those XML files. But first we need to tell you a story.

The story about PojoSR that became Felix Connector

When using OSGi Blueprint with Camel, you must run your Camel applications in an OSGi server such as Apache Karaf/ServiceMix or JBoss Fuse. Many years ago, the only way to test your Camel applications was to deploy them into such an OSGi server.

This changed in 2011 when Karl Pauls created PojoSR, an OSGi Lite container. PojoSR runs without an OSGi framework and uses a flat class loader. Instead, PojoSR bootstraps a simulated OSGi environment: you colocate

your unit test and Camel dependencies and run in the same flat class loader. In other words, it runs in the same JVM locally and therefore starts up fast. You'll also be pleased to know that because it all runs in the same JVM, Java debugging is much easier: set a breakpoint in your IDE editor and click the debug button—no need for remote Java debugging. But the flip side is that the more OSGi needs you have, the bigger the chance that PojoSR has reached its limits. When you have such needs, you can turn to using an alternative testing framework called Pax Exam or Arquillian (covered in section 9.6). PojoSR has since become part of Apache Felix as the Felix Connect project.

To start unit-testing Camel with OSGi Blueprint XML files, you add the following dependency to your Maven pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test-blueprint</artifactId>
  <version>2.20.1</version>
</dependency>
```

To keep this example basic, you'll use the following OSGi Blueprint XML file with the file copier Camel route, as shown in the following listing.

Listing 9.9 An OSGi Blueprint XML file with a basic Camel route (firststep.xml)

```
<blueprint
  xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://www.osgi.org/xmlns/blueprint/v1.0.0
    https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

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

```
<route>
    <from uri="file://target/inbox"/>
    <to uri="file://target/outbox"/>
</route>
</camelContext>
</blueprint>
```

To unit test this route, you extend the class `CamelBlueprintTestSupport` and write the unit test almost like that in the previous section. The following listing shows the code.

[Listing 9.10](#) OSGi Blueprint-based unit test

```
public class BlueprintFirstTest extends
CamelBlueprintTestSupport { ①
```

①

Test class must extend `CamelBlueprintTestSupport`

```
@Override
protected String getBlueprintDescriptor() {
    return "camelinaction/firststep.xml"; ②
```

②

Loads the OSGi Blueprint XML file

```
}
```

```
@Test
```

```
public void testMoveFile() throws Exception { ③
```

③

Test method is similar to previous examples

```
template.sendBodyAndHeader("file://target/inbox",
    "Hello World", Exchange.FILE_NAME,
"hello.txt");

Thread.sleep(2000);
```

```
        File target = new File("target/outbox/hello.txt");
        assertTrue("File not moved", target.exists());
        String content = context.getTypeConverter()
                            .convertTo(String.class,
target);
        assertEquals("Hello World", content);
    }
}
```

As you can see, the `BlueprintFirstTest` class extends `CamelBlueprintTestSupport`, which is required ❶. Then you override the method `getBlueprintDescriptor`, where you return the location in the classpath of the OSGi Blueprint XML file ❷ (you can specify multiple files by using a comma to separate the locations). The rest of the test method is similar to previous examples ❸.

You can try this example, located in the `chapter9/blueprint-xml` directory, using the following Maven goal:

```
mvn test -Dtest=BlueprintFirstTest
```

We have one more trick to show you with OSGi Blueprint. When using OSGi, you can compose your applications into separate bundles and services. You may compose shared services or let different teams build different services. Then, all together, these services can be installed in the same OSGi servers and just work together—famous last words.

MOCKING OSGI SERVICES

If you're on one of these teams, how can you unit-test your Camel applications with shared services across your Camel application? What you can do with `camel-test-blueprint` is mock these services and register them manually in the OSGi service registry.

Suppose your Camel Blueprint route depends on an OSGi service, as shown here:

```
<reference id="auditService"
interface="camelaction.AuditService"/>
```

❶

1

Reference to existing OSGi service

```
<camelContext  
xmlns="http://camel.apache.org/schema/blueprint">  
    <route>  
        <from uri="file:///target/inbox"/>  
        <bean ref="auditService"/> 2
```

2

Calling the OSGi service from Camel route

```
        <to uri="file:///target/outbox"/>  
    </route>  
</camelContext>
```

The `<reference>` 1 is a reference to an existing OSGi service in the OSGi service registry that's identified only by the interface that must be of type `camelinaction.AuditService`. The service is then used in the Camel route 2.

If you attempt to unit-test this Camel route with `camel-test-blueprint`, the test will fail with an OSGi Blueprint error stating a time-out error waiting for the service to be available.

Yes, this is a simple example, but imagine that your Camel applications are using any number of OSGi services that are built by other teams. How can you build and run your Camel unit tests?

What you can do is implement mocks for these services and register these fake services in your unit test. For example, you could implement a mocked service that when called will send the Camel Exchange to a mock endpoint, as shown here:

```
public class MockAuditService implements AuditService {  
    public void audit(Exchange exchange) {  
        exchange.getContext().createProducerTemplate()  
            .send("mock:audit", exchange);  
    }  
}
```

Then you can register `MockAuditService` as the fake service in the OSGi service registry as part of your unit test using the `addServicesOnStartup` method:

```
protected void addServicesOnStartup(Map<String,
                                     KeyValueHolder<Object,
Dictionary>> services) {
    MockAuditService mock = new MockAuditService()
```

Creates mock of the AuditService

```
    services.put(AuditService.class.getName(),
asService(mock, null));
```

Registers the mock into the OSGi ServiceRegistry

```
}
```

You can find this example with the source code in the `chapter9/blueprint-xml-service` directory, and you can try the example by using the following Maven goal:

```
mvn test
```

We'll now move on to using Camel with CDI. You'll see how to start testing Camel running in a CDI container such as JBoss Weld and then using a Java EE application server such as WildFly.

9.2.5 CAMEL TESTING WITH CDI

Camel has great integration with Contexts and Dependency Injection (CDI) that allows you to build Camel routes using Java code.

TIP Chapter 7 covered CDI. You'll encounter CDI again in chapter 15.

To start unit-testing Camel with CDI, you add the following

dependency to your Maven pom.xml file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-test-cdi</artifactId>
    <version>2.20.1</version>
</dependency>
```

As usual we'll use a basic Camel example to show you the ropes to get on board testing with CDI. The Camel route used for testing is a file copier example:

`@Singleton` 1

Annotates the class with `@Singleton` to make Camel discover the class easily

```
public class FileMoveRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("file:target/inbox")
            .to("file:target/outbox");
    }
}
```

The class is annotated with `@Singleton`, which allows Camel to discover the instance on startup and automatically include the route.

To unit-test this Camel route, you don't extend a base class but use `CamelCdiRunner`, a JUnit runner, as shown next.

[Listing 9.11](#) Unit testing Camel CDI application

`@RunWith(CamelCdiRunner.class)` 1

1

Uses `@RunWith` with the Camel CDI runner

```
public class FirstTest {
    @Inject
```

```
private CamelContext context; ②
```

②

Dependency injects CamelContext and ProducerTemplate

```
@Inject @Uri("file:target/inbox")  
private ProducerTemplate template; ②
```

```
@Test  
public void testMoveFile() throws Exception { ③
```

③

The unit test method is identical to previous unit tests covered previously.

```
template.sendBodyAndHeader("Hello World",  
                           Exchange.FILE_NAME,  
                           "hello.txt");  
  
Thread.sleep(2000);  
  
File target = new File("target/outbox/hello.txt");  
assertTrue("File not moved", target.exists());  
String content = context.getTypeConverter()  
                  .convertTo(String.class,  
target);  
assertEquals("Hello World", content);  
}  
  
}
```

To run the test as JUnit, you use `CamelCdiRunner` as `@RunWith` **①**. Because the `FirstTest` class doesn't extend a base class, you need to dependency-inject the resources used during testing such as `CamelContext` and `ProducerTemplate` **②**. The test method is implemented similarly to what's been covered before in this section **③**.

You can try this example, which is provided with the source code in the `chapter9/cdi` directory, using the following Maven goal:

```
mvn test -Dtest=FirstTest
```

The JBoss Weld CDI container doesn't offer much power besides being able to run a CDI application. If you're looking for a more powerful and better container, take a look at WildFly Swarm.

9.2.6 CAMEL TESTING WITH WILDFLY SWARM

You first encountered WildFly Swarm in this book in chapter 7. You may think WildFly Swarm is similar to Spring Boot but for the Java EE specification. Therefore, it's best to run Camel with CDI on WildFly Swarm. This time, you'll use a simple Camel route that routes a message between two in-memory SEDA queues, as shown here:

`@Singleton`

Annotates the class with `@Singleton` to make CDI discover the class easily

```
public class SedaRoute extends RouteBuilder {  
    @Override  
    public void configure() throws Exception {  
        from("seda:inbox")  
            .to("seda:outbox");  
    }  
}
```

WildFly Swarm comes with powerful test capabilities using Arquillian. Setting up Arquillian is fairly trivial. First, you need to add the WildFly Swarm Arquillian dependency to the Maven pom.xml file:

```
<dependency>  
    <groupId>org.wildfly.swarm</groupId>  
    <artifactId>arquillian</artifactId>  
    <scope>test</scope>  
</dependency>
```

Then all that's left is to write the unit test, shown in the following listing.

[Listing 9.12](#) Testing Camel using Arquillian on WildFly Swarm

```
@RunWith(Arquillian.class)
```

1

1

Sets up unit test to run with Arquillian

```
@DefaultDeployment(type = DefaultDeployment.Type.JAR)
```

2

2

Uses the default deployment

```
public class WildFlySwarmCamelTest {
```

```
    @Inject 3
```

3

Injects CamelContext

```
    private CamelContext camelContext; 3
```

```
    @Inject @Uri("seda:inbox") 4
```

4

Injects ProducerTemplate to send to seda:inbox

```
    private ProducerTemplate template; 4
```

```
    @Test
```

```
    public void testSeda() throws Exception {  
        template.sendBody("Hello Swarm");
```

```
        ConsumerTemplate consumer =  
        camelContext.createConsumerTemplate();  
        Object body = consumer.receiveBody("seda:outbox",  
        5000); 5
```

5

Receives message from seda:outbox

```
        assertEquals("Hello Swarm", body);
```

```
}
```

The class `WildFlySwarmCamelTest` has been annotated with JUnit `@RunWith(Arquillian)` ❶ to set up and run the unit test using Arquillian. Because your Camel application is running inside WildFly Swarm, you've annotated the class with `@DefaultDeployment` ❷, which instructs Arquillian to include your entire application together with WildFly Swarm. Notice that you specify `type = DefaultDeployment.Type.JAR`, which is required when using WildFly Swarm in JAR packaging mode, instead of the default WAR packaging mode. You'll do this most often with WildFly Swarm because it's intended for microservices, where you deploy and run only one application in the server. But Arquillian is capable of deploying only what you've instructed when you run in traditional application servers, which you'll try in the following section.

The test uses CDI to inject `camelContext` ❸ and `ProducerTemplate` ❹, which you use in the test method. The producer is used to send a message to the `seda:inbox` endpoint followed by the `consumerTemplate` ❺ to retrieve the message from the `seda:outbox` endpoint.

The consumer is using a five-second time-out. If you don't specify a time-out, the consumer will wait until a message is received. Because this is a unit test, you want to fail instead if no message is routed by Camel. This will happen in case of a time-out because the message body would be `null` and fail the equals check at the end of the test.

You can try this example, which is part of the source code in the `chapter9/wildfly-swarm` directory, by using the following Maven goal:

```
mvn test -Dtest=WildFlySwarmCamelTest
```

You can also run a Camel application in Java EE application servers such as WildFly. But how do you test with Camel and WildFly?

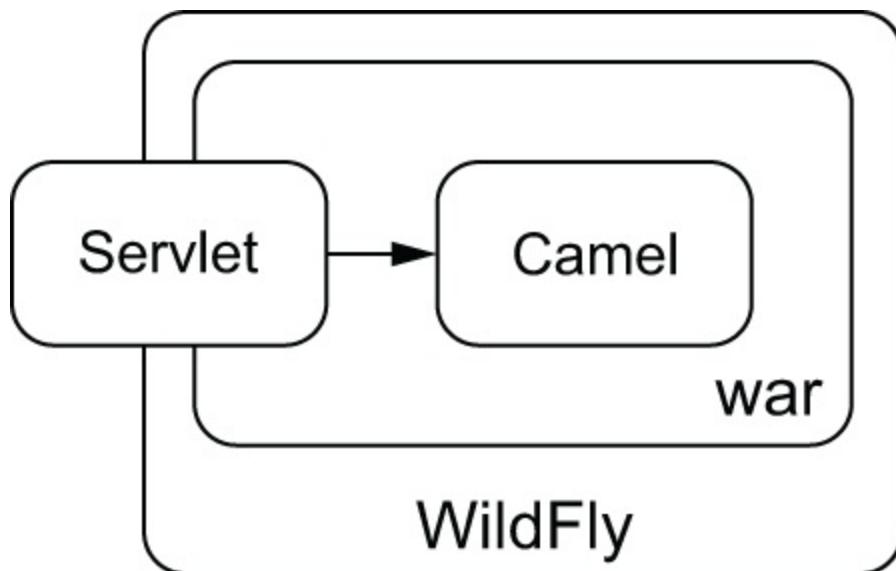
9.2.7 CAMEL TESTING WITH WILDFLY

There are several ways of testing Camel with WildFly. The simplest form is a regular unit test that runs standalone outside WildFly. This unit test would be much the same as those we've already covered:

- Testing Camel with Java routes (covered in section 9.1.2)
- Testing Camel with Spring XML routes (covered in section 9.2.1)

In this section, we raise the bar to talk about how to test Camel with WildFly when the test runs inside the WildFly application server. This allows you to do integration tests with *the real thing* (the application server).

You'll build a simple example that uses Java EE technology together with Camel running in a WildFly application server. The example exposes a servlet that calls a Camel route and returns a response. [Figure 9.3](#) illustrates how the example is deployed as a WAR deployment running inside the WildFly server.



[Figure 9.3](#) A WAR deployment using servlet and Camel running in WildFly application server

The servlet is implemented as shown in the following listing.

[Listing 9.13](#) A hello Servlet that calls a Camel route

```
@WebServlet(name = "HelloServlet", urlPatterns = {"/*"},  
loadOnStartup = 1)  
public class HelloServlet extends HttpServlet { ①
```

①

`@WebServlet` to make this class act as a servlet

```
@Inject  
private ProducerTemplate producer; ②
```

②

Dependency injects a Camel ProducerTemplate using CDI injection

```
protected void doGet(HttpServletRequest req,  
                      HttpServletResponse res) throws  
IOException {  
    String name = req.getParameter("name");  
  
    String s = producer.requestBody("direct:hello", name,  
String.class); ③
```

③

Calls the Camel route to retrieve an output to use as response

```
    res.getOutputStream().print(s);  
}  
}
```

The servlet is implemented in the `HelloServlet` class. By annotating the class with `@WebServlet` ①, you don't have to register the servlet by using the WEB-INF/web.xml file. The servlet integrates with Camel by using CDI to inject `ProducerTemplate` ②. The servlet implements the `doGet` method that handles HTTP GET methods. In this method, the query parameter name is extracted and used as message body when calling the Camel route ③ to obtain a response message from Camel.

The Camel route is a simple route that transforms a message:

```
@ContextName("helloCamel")
public class HelloRoute extends RouteBuilder {

    public void configure() throws Exception {
        from("direct:hello")
            .transform().simple("Hello ${body}");
    }
}
```

This example is packaged as a WAR deployment unit that can be deployed to any Java EE web server such as Apache Tomcat or WildFly. We'll now show you how to test your deployments running inside WildFly.

To be able to test this, you use Arquillian, which allows you to write unit or integration tests that can deploy your application and tests together and run the tests inside a running WildFly server:

Arquillian provides a component model for integration tests, which includes dependency injection and container lifecycle management. Instead of managing a runtime in your test, Arquillian brings your test to the runtime.—Arquillian website, <http://arquillian.org/modules/core-platform/>

TESTING WILDFLY USING ARQUILLIAN

To use Arquillian to perform integration tests using WildFly, you need to add the arquillian-bom and shrinkwrap-resolver-bom BOMs to your Maven pom.xml file. In addition, you need to add the following three dependencies:

- arquillian-junit-container—Provides the Arquillian JUnit runner
- shrinkwrap-resolver-impl-maven—To resolve dependencies using Maven
- wildfly-arquillian-container-managed—Deploys and runs the tests in WildFly

You can find the exact details in the pom.xml file in the example

source code, located in the chapter9/wildfly directory.

Writing an Arquillian-based integration test is similar to writing a unit test. The following listing shows what it takes.

Listing 9.14 System test that tests your servlet and Camel application running in WildFly

```
@RunWith(Arquillian.class) ①
```

①

JUnit runner to run using Arquillian

```
public class FirstWildFlyIT {
```

```
    @Inject
```

```
    CamelContext camelContext; ②
```

②

Dependency injects CamelContext

```
@Deployment
```

```
public static WebArchive createDeployment() {
```

```
    File[] files = Maven.resolver()
```

```
        .loadPomFromFile("pom.xml")
```

```
        .importRuntimeDependencies()
```

```
    .resolve().withTransitivity().asFile(); ③
```

③

ShrinkWrap to resolve all the runtime dependencies from the Maven pom.xml file

```
WebArchive archive =  
ShrinkWrap.create(WebArchive.class, ④)
```

④

Creates WebArchive deployment using ShrinkWrap

```
        "mycamel-  
wildfly.war");  
        archive.addAsWebInfResource(EmptyAsset.INSTANCE,  
"beans.xml");  
        archive.addPackages(true, "camelinaction"); 5
```

5

Includes the project classes

```
archive.addAsLibraries(files); 6
```

6

Adds all the Maven dependencies as libraries in the deployment unit

```
    return archive;  
}  
  
@Test  
public void testHello() throws Exception { 7
```

7

The test method that calls the Camel route

```
String out = camelContext.createProducerTemplate()  
                    .requestBody("direct:hello", "Donald",  
String.class);  
        assertEquals("Hello Donald", out);  
    }  
}
```

The test class `FirstWildFlyIT` uses the Arquillian JUnit runner **1** that lets Arquillian be in charge of the tests. Then you use CDI to inject `CamelContext` into the test class **2**. To include all needed dependencies in the deployment unit, you use ShrinkWrap Maven resolver **3** that includes all the runtime dependencies and their transitive dependencies. The deployment unit is then created **4** by using ShrinkWrap as a WAR deployment with the name `mycamel-wildfly.war`. It's important that you use the correct name of the WAR file as configured in the Maven `pom.xml` file:

```
<finalName>mycamel-wildfly</finalName>
```

The method `addPackages` ❸ includes all the classes from the root package `camelinaction` and all subpackages. This ensures that all the classes of this project are included in the deployment unit. The method `addAsLibraries` ❹ adds all the dependencies ❻ as JARs to the deployment unit. The unit-test method is a simple test that calls the Camel route ❼. You could've chosen to perform an HTTP call to the servlet instead, but then you'd need to add an HTTP library. You cheat a bit and just call the Camel route.

Okay, that's a mouthful to take in; are you ready for testing? No, you still have two tasks to do: set up Arquillian and install WildFly.

You configure Arquillian in the `arquillian.xml` file, which you put in the `src/test/resources` directory. Its content is shown here:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<arquillian xmlns="http://jboss.org/schema/arquillian"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
             xsi:schemaLocation="
               http://jboss.org/schema/arquillian
               http://jboss.org/schema/arquillian_1_0.xsd">
  <engine>
    <property
      name="deploymentExportPath">target</property> ❶

```

❶

To use the target directory during deployment and testing

```
  </engine>
  <container qualifier="managed" default="true"> ❷

```

❷

Uses a managed WildFly container

```
  <configuration>
    <property name="jbossHome">${env.JBOSS_HOME}</property> ❸

```

③

The location of the JBoss WildFly installation

```
<property  
name="serverConfig">standalone.xml</property>
```

④

④

To use the standalone.xml configuration with WildFly

```
<property  
name="allowConnectingToRunningServer">true</property>  
  </configuration>  
  </container>  
</arquillian>
```

The arquillian.xml configuration file is fairly easy to comprehend. At first, you configure to use the target directory ① as a work directory so it's easy to clean up using a Maven clean goal. You use Arquillian container testing in managed mode ②. This means an existing installation of WildFly is used, so you need to configure the directory where WildFly is installed. The recommended way is to use the JBOSS_HOME environment variable ③. Then you tell Arquillian to use the default WildFly standalone.xml configuration file when running WildFly ④.

TIP What you learn here about using Arquillian for testing with WildFly is similar in principle to using other containers such as Apache Tomcat or Jetty.

The last piece of the puzzle is to install WildFly.

INSTALLING WILDFLY

You can download WildFly from <http://wildfly.org>. The installation of WildFly is a matter of unzipping the downloaded file and setting up the JBOSS_HOME environment variable.

We have installed WildFly 11 in the /opt directory and set up

the `JBOSS_HOME` environment as follows:

```
export JBOSS_HOME=/opt/wildfly-11.0.0.Final
```

After this is done, you're ready for running the integration tests, which is as simple as running a Maven test. We've included this example in the source code in the `chapter9/wildfly` directory, and you can try the example by using the following:

```
mvn integration-test -P wildfly
```

We took our time to walk you through setting up a project for system tests with Arquillian that runs in WildFly. What you've learned here will come in handy in section 9.6, which provides more information about system tests using various test libraries such as Arquillian and Pax Exam.

You've now seen the Camel Test Kit and learned to use its JUnit extension to write your first unit tests with various runtime platforms such as standalone Java, Spring Boot, OSGi, CDI, and WildFly. Camel helps a lot when working with files, but things get more complex when you use more protocols—especially complex ones such as Java Message Service (JMS) messaging. Testing an application that uses many protocols has always been challenging.

This is why mocks were invented. Using mocks, you can simulate real components and reduce the number of variables in your tests. Mock components are the topic of the next section.

Now is a good time to take a little break. We're changing the scene from running Camel tests on various runtimes to using the Camel mock component, which is useful in your unit tests.

9.3 Using the mock component

The *mock component* is a cornerstone when testing with Camel; it makes testing much easier. In much the same way that a car designer uses a crash-test dummy to simulate vehicle impact on humans, the mock component is used to simulate real

components in a controlled way.

Mock components are useful in several situations:

- When the real component doesn't yet exist or isn't reachable in the development and test phases. For example, if you have access to the component in only preproduction and production phases.
- When the real component is slow or requires much effort to set up and initialize, such as a database.
- When you'd have to incorporate special logic into the real component for testing purposes, which isn't practical or possible.
- When the component returns nondeterministic results, such as the current time, which would make it difficult to unit-test at any given time of day.
- When you need to simulate errors caused by network problems or faults from the real component.

Without the mock component, your only option is to test using the real component, which is usually much harder. You may already have used mocking before—many frameworks blend in well with testing frameworks such as JUnit.

Camel takes testing seriously, and the mock component was included in the first release of Camel. The fact that it resides in the camel-core JAR indicates its importance. The mock component is used rigorously in unit-testing Camel itself.

In this section, you'll look at how to use the mock component in unit tests and how to add mocking to existing unit tests. Then you'll spend some time using mocks to set expectations to verify test results, as this is where the mock component excels.

Let's get started.

9.3.1 INTRODUCING THE MOCK COMPONENT

Figure 9.4 illustrates the three basic steps of testing.

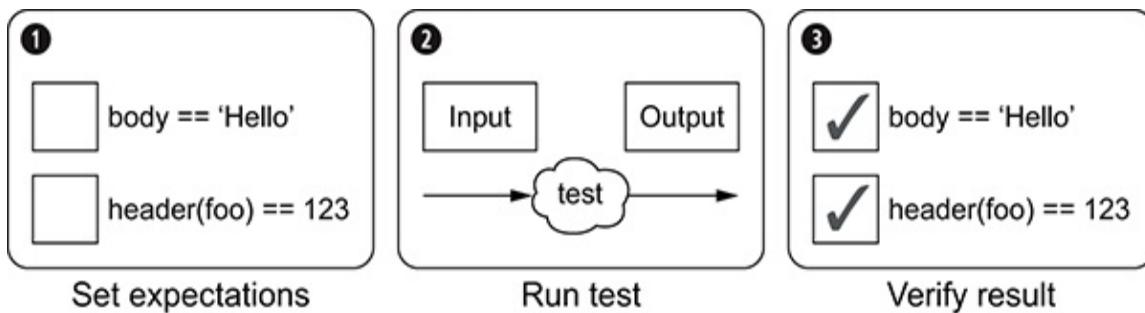


Figure 9.4 Three steps for testing: set expectations, run the test, and verify the result

Before the test is started, you set the expectations of what should happen ①. Then you run the test ②. Finally, you verify the outcome of the test against the expectations ③. The Camel mock component allows you to easily implement these steps when testing Camel applications. On the mock endpoints, you can set expectations that are used to verify the test results when the test completes.

Mock components can verify a rich variety of expectations, such as the following:

- The correct number of messages are received on each endpoint.
- The messages arrive in the correct order.
- The correct payloads are received.
- The test ran within the expected time period.

Mock components allow you to configure coarse- and fine-grained expectations and simulate errors such as network failures. Let's get started and try using the mock component.

9.3.2 UNIT-TESTING WITH THE MOCK COMPONENT

As you learn how to use the mock component, you'll use the following basic route to keep things simple:

```
from("jms:topic:quote").to("mock:quote");
```

This route will consume messages from a JMS topic, named

quote, and route the messages to a mock endpoint with the name quote.

The mock endpoint is implemented in Camel as the `org.apache.camel.component.mock.MockEndpoint` class; it provides a large number of methods for setting expectations. [Table 9.3](#) lists the most commonly used methods on the mock endpoint.

Table 9.3 Commonly used methods in the MockEndpoint class

Method	Description
<code>expectedMessageCount(intcount)</code>	Specifies the expected number of messages arriving at the endpoint
<code>expectedMinimumMessageCount(intcount)</code>	Specifies the expected minimum number of messages arriving at the endpoint
<code>expectedBodiesReceived(Object...bodies)</code>	Specifies the expected message bodies and their order arriving at the endpoint
<code>expectedBodiesReceivedInAnyOrder(Object...bodies)</code>	Specifies the expected message bodies arriving at the endpoint—ordering doesn't matter
<code>assertIsSatisfied()</code>	Validates that all expectations set on the endpoint are satisfied

The `expectedMessageCount` method is exactly what you need to set the expectation that one message should arrive at the `mock:quote` endpoint. You can do this as shown in the following listing.

Listing 9.15 Using MockEndpoint in unit testing

```
public class FirstMockTest extends CamelTestSupport {  
  
    protected RouteBuilder createRouteBuilder() throws  
Exception {  
        return new RouteBuilder() {  
            public void configure() throws Exception {  
                from("jms:topic:quote").to("mock:quote");  
            }  
        };  
    }  
}
```

```
@Test  
public void testQuote() throws Exception {  
    MockEndpoint quote = getMockEndpoint("mock:quote");  
    quote.expectedMessageCount(1); 1
```

1

Expects one message

```
template.sendBody("jms:topic:quote", "Camel rocks");  
quote.assertIsSatisfied(); 2
```

2

Verifies expectations

```
}
```

To obtain the `MockEndpoint`, you use the `getMockEndpoint` method from the `CamelTestSupport` class. Then you set your expectations—in this case, you expect one message to arrive **1**. You start the test by sending a message to the JMS topic, and the mock endpoint verifies whether the expectations were met by using the `assertIsSatisfied` method **2**. If a single expectation fails, Camel throws a `java.lang.AssertionError` stating the failure.

You can compare what happens in [listing 9.14](#) to what you saw in [figure 9.4](#): you set expectations, run the test, and verify the results. It can't get any simpler than that.

NOTE By default, the `assertIsSatisfied` method runs for 10 seconds before timing out. You can change the wait time with the `setResultWaitTime(longtimeInMillis)` method if you have unit tests that run for a long time. But it's easier to specify the time-out directly as a parameter to the `assert` method:
`assertIsSatisfied(5, TimeUnit.SECONDS);`

REPLACING JMS WITH STUB

[Listing 9.14](#) uses JMS, but for now let's keep things simple by using the stub component to simulate JMS. (You'll look at testing JMS with ActiveMQ in section 9.4.) The stub component is essentially the SEDA component with parameter validation turned off. This makes it possible to stub any Camel endpoint by prefixing the endpoint with `stub:`. In this example, you have the endpoint `jms:topic:quote`, which can be stubbed as `stub:jms:topic:quote`. The route will be as simple as this:

```
from("stub:jms:topic:quote").to("mock:quote");
```

The producer can then send a message to the queue by using the following:

```
template.sendBody("stub:jms:topic:quote", "Camel rocks");
```

We've provided this example in the source code, in the `chapter9/mock` directory; you can try the example by using the following Maven goals:

```
mvn test -Dtest=FirstMockTest  
mvn test -Dtest=SpringFirstMockTest
```

You may have noticed in [listing 9.15](#) that the expectation is coarse-grained in the sense that you expect only one message to arrive. You don't specify anything about the message's content or other characteristics, so you don't know whether the message that arrives is the same "Camel rocks" message that was sent. The next section covers how to test this.

9.3.3 VERIFYING THAT THE CORRECT MESSAGE ARRIVES

The `expectedMessageCount` method can be used to verify only that a certain number of messages arrive. It doesn't dictate anything about the content of the message. Let's improve the unit test in [listing 9.15](#) so that it expects the message being sent to match the message that arrives at the mock endpoint.

You can do this by using the `expectedBodiesReceived` method,

as follows:

```
@Test
public void testQuote() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:quote");
    mock.expectedBodiesReceived("Camel rocks");

    template.sendBody("stub:jms:topic:quote", "Camel rocks");

    mock.assertIsSatisfied();
}
```

This is intuitive and easy to understand, but the method states *bodies*, plural, as if there could be more bodies. Camel does support expectations of multiple messages, so you could send in two messages. Here's a revised version of the test:

```
@Test
public void testQuotes() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:quote");
    mock.expectedBodiesReceived("Camel rocks", "Hello
Camel");

    template.sendBody("stub:jms:topic:quote", "Camel rocks");
    template.sendBody("stub:jms:topic:quote", "Hello Camel");

    mock.assertIsSatisfied();
}
```

Camel now expects two messages to arrive in the specified order. Camel will fail the test if the “Hello Camel” message arrives before the “Camel rocks” message.

If the order doesn't matter, you can use the `expectedBodiesReceivedInAnyOrder` method instead, like this:

```
mock.expectedBodiesReceivedInAnyOrder("Camel rocks", "Hello
Camel");
```

It could hardly be any easier than that.

But if you expect a much larger number of messages to arrive, the number of bodies you pass in as an argument will be large. How can you do that? The answer is to use a `List` containing the expected bodies as a parameter:

```
List bodies = ...  
mock.expectedBodiesReceived(bodies);
```

This example is available in the source code, in the chapter9/mock directory; you can try the example by using the following Maven goals:

```
mvn test -Dtest=FirstMockTest  
mvn test -Dtest=SpringFirstMockTest
```

The mock component has many other features. Let's continue by exploring how to use expressions to set fine-grained expectations.

9.3.4 USING EXPRESSIONS WITH MOCKS

Suppose you want to set an expectation that a message should contain the word *Camel* in its content. The following listing shows one way of doing this.

Listing 9.16 Using expressions with MockEndpoint to set expectations

```
@Test  
public void testIsCamelMessage() throws Exception {  
    MockEndpoint mock = getMockEndpoint("mock:quote");  
    mock.expectedMessageCount(2); 1
```

1

Expects two messages

```
template.sendBody("stub:jms:topic:quote", "Hello Camel");  
template.sendBody("stub:jms:topic:quote", "Camel rocks");  
  
assertMockEndpointsSatisfied(); 2
```

2

Verifies two messages received

```
List<Exchange> list = mock.getReceivedExchanges();
```

③

③

Verifies “Camel” is in received messages

```
String body1 =
list.get(0).getIn().getBody(String.class); ③
String body2 =
list.get(1).getIn().getBody(String.class); ③
assertTrue(body1.contains("Camel")); ③
assertTrue(body2.contains("Camel")); ③
}
```

First, you set up your expectation that the `mock:quote` endpoint will receive two messages ①. You then send two messages to the JMS topic to start the test. Next, you assert that the mock received the two messages by using the `assertMockEndpointsSatisfied` method ②, which is a one-stop method for asserting all mocks. This method is more convenient to use than having to invoke the `assertIsSatisfied` method on *every* mock endpoint you may have in use.

At this point, you can use the `getReceivedExchanges` method to access all the exchanges the `mock:quote` endpoint has received ③. You use this method to get hold of the two received message bodies so you can assert that they contain the word *Camel*.

At first you may think it a bit odd to define expectations in two places—before and after the test has run. Is it not possible to define the expectations in one place, such as before you run the test? Yes, it is, and this is where Camel expressions come into the game.

NOTE The `getReceivedExchanges` method still has its merits. It allows you to work with the exchanges directly, giving you the ability to do whatever you want with them.

Table 9.4 lists some additional `MockEndpoint` methods that let

you use expressions to set expectations.

Table 9.4 Expression-based methods commonly used on MockEndpoint

Method	Description
message(int index)	Defines an expectation on the <i>n</i> th message received
allMessages()	Defines an expectation on all messages received
expectsAscending(Expressionexpression)	Expects messages to arrive in ascending order
expectsDescending(Expressionexpression)	Expects messages to arrive in descending order
expectsDuplicates(Expressionexpression)	Expects duplicate messages
expectsNoDuplicates(Expressionexpression)	Expects no duplicate messages
expects(Runablerunable)	Defines a custom expectation

You can use the `message` method to improve the unit test in [listing 9.14](#) and group all your expectations together, as shown here:

```
@Test
public void testIsCamelMessage() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:quote");
    mock.expectedMessageCount(2);
    mock.message(0).body().contains("Camel");
    mock.message(1).body().contains("Camel");

    template.sendBody("stub:jms:topic:quote", "Hello Camel");
    template.sendBody("stub:jms:topic:quote", "Camel rocks");

    assertMockEndpointsSatisfied();
}
```

Notice that you can use the `message(int index)` method to set an expectation that the body of the message should contain the word *Camel*. Instead of doing this for each message based on its

index, you can use the `allMessages` method to set the same expectation for all messages:

```
mock.allMessages().body().contains("Camel");
```

So far, you've seen expectations based on only the message body, but what if you want to set an expectation based on a header? That's easy—you use `header(name)`, as follows:

```
mock.message(0).header("JMSPriority").isEqualTo(4);
```

You probably noticed the `contains` and `isEqualTo` methods used in the preceding couple of code snippets. They're builder methods used to create predicates for expectations. [Table 9.5](#) lists all the builder methods available.

Table 9.5 Builder methods for creating predicates to be used as expectations

Method	Description
<code>contains(Objectvalue)</code>	Sets an expectation that the message body contains the given value.
<code>isInstanceOf(ClassType)</code>	Sets an expectation that the message body is an instance of the given type.
<code>startsWith(Objectvalue)</code>	Sets an expectation that the message body starts with the given value.
<code>endsWith(Objectvalue)</code>	Sets an expectation that the message body ends with the given value.
<code>in(Object...values)</code>	Sets an expectation that the message body is equal to any of the given values.
<code>isEqualTo(Objectvalue)</code>	Sets an expectation that the message body is equal to the given value.
<code>isNotEqualTo(Objectvalue)</code>	Sets an expectation that the message body isn't equal to the given value.
<code>isGreaterT</code>	Sets an expectation that the message body is greater than the given

han(Object value)	value.
isGreaterT hanOrEqual (Objectval ue)	Sets an expectation that the message body is greater than or equal to the given value.
isLessThan (Objectval ue)	Sets an expectation that the message body is less than the given value.
isLessThan OrEqual(Ob jectvalue)	Sets an expectation that the message body is less than or equal to the given value.
isNull(Obj ectvalue)	Sets an expectation that the message body is <code>null</code> .
isNotNull(Objectvalu e)	Sets an expectation that the message body isn't <code>null</code> .
matches(Expression expression)	Sets an expectation that the message body matches the given expression. The expression can be any of the supported Camel expression languages such as Simple, Groovy, SpEL, or MVEL.
regex(Stringpattern)	Sets an expectation that the message body matches the given regular expression.

At first, it may seem odd that the methods in table 9.5 often use `Object` as the parameter type. Why not a specialized type such as `String`? That's because of Camel's strong type-converter mechanism, which allows you to compare apples to oranges; Camel can regard both of them as fruit and evaluate them accordingly. You can compare strings with numeric values without having to worry about type mismatches, as illustrated by the following two code lines:

```
mock.message(0).header("JMSPriority").isEqualTo(4);
mock.message(0).header("JMSPriority").isEqualTo("4");
```

Now suppose you want to create an expectation that all messages contain the word *Camel* and end with a period. You could use a regular expression to set this in a single expectation:

```
mock.allMessages().body().regex(".*Camel.*\\\"");
```

This will work, but Camel allows you to enter multiple expectations, so instead of using the `regex` method, you can create a more readable solution:

```
mock.allMessages().body().contains("Camel");
mock.allMessages().body().endsWith(".");
```

The source code includes examples that you can try by running the following Maven command from the chapter9/mock directory:

```
mvn test -Dtest=SecondMockTest
mvn test -Dtest=SpringSecondMockTest
```

You've learned a lot about how to set expectations, including fine-grained ones using the builder methods listed in table 9.5.

Now let's get back to the mock components and learn about using mocks to simulate real components. This is useful when the real component isn't available or isn't reachable from a local or test environment.

9.3.5 USING MOCKS TO SIMULATE REAL COMPONENTS

Suppose you have a route like the following one, in which you expose an HTTP service using Jetty so clients can obtain an order status:

```
from("jetty:http://web.rider.com/service/order")
    .process(new OrderQueryProcessor())
    .to("netty4:tcp://miranda.rider.com:8123?textline=true")
    .process(new OrderResponseProcessor());
```

Clients send an HTTP GET, with the order ID as a query parameter, to the <http://web.rider.com/service/order> URL. Camel will use the `OrderQueryProcessor` to transform the message into a format that the Rider Auto Parts mainframe (named Miranda) understands. The message is then sent to Miranda using TCP, and Camel waits for the reply to come back.

The reply message is then processed using the `OrderResponseProcessor` before it's returned to the HTTP client.

Now suppose you're asked to write a unit test to verify that clients can obtain the order status. The challenge is that you don't have access to Miranda, which contains the order status. You're asked to simulate this server by replying with a canned response.

Camel provides the two methods listed in table 9.6 to help simulate a real component.

Table 9.6 Methods to control responses when simulating a real component

Method	Description
<code>whenAnyExchangeReceived (Processorprocessor)</code>	Uses a custom processor to set a canned reply
<code>whenExchangeReceived (intindex, Processorprocessor)</code>	Uses a custom processor to set a canned reply when the <i>n</i> th message is received

You can simulate a real endpoint by mocking it with the mock component and using the methods in table 9.6 to control the reply. To do this, you need to replace the endpoint in the route with the mocked endpoint, which is done by replacing it with `mock:miranda`. Because you want to run the unit test locally, you also need to change the HTTP hostname to `localhost`, allowing you to run the test locally on your own laptop:

```
from("jetty:http://localhost:9080/service/order")
    .process(new OrderQueryProcessor())
    .to("mock:miranda")
    .process(new OrderResponseProcessor());
```

The unit test that uses the preceding route follows.

Listing 9.17 Simulating a real component by using a mock endpoint

```
public class MirandaTest extends CamelTestSupport {
```

```
@Test
public void testMiranda() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:miranda");
    mock.expectedBodiesReceived("ID=123");
    mock.whenAnyExchangeReceived(e ->
        e.getIn().setBody("ID=123, STATUS=IN PROGRESS")) ①
```

①

Returns canned response

```
);

String url = "http://localhost:9080/service/order?
id=123";

String out = template.requestBody(url, null,
String.class);
assertEquals("IN PROGRESS", out); ②
```

②

Verifies expected reply

```
    assertMockEndpointsSatisfied();
}

protected RouteBuilder createRouteBuilder() throws
Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("jetty://http://localhost:9080/service/order")
                .transform().message(m -> "ID=" +
m.getHeader("id")) ③
```

③

Transforms to format understood by Miranda

```
.to("mock:miranda")
.transform().body(String.class,
b -> StringHelper.after(b,
"STATUS=")); ④
```

④

Transforms to response format

```
        }  
    };  
}  
}
```

In the `testMiranda` method, you obtain the `mock:miranda` endpoint, which is the mock that simulates the Miranda server, and you set an expectation that the input message contains the body "ID=123". To return a canned reply, you use the `whenAnyExchangeReceived` method ❶, which allows you to use a custom processor to set the canned response. This response is set to be `ID=123, STATUS=IN PROGRESS`.

Then you start the unit test by sending a message to the `http://localhost:9080/service/order?id=123` endpoint; the message is an HTTP GET using the `requestBody` method from the `template` instance. You then assert that the reply is IN PROGRESS by using the regular JUnit `assertEquals` method ❷.

Instead of using two processors to transform the data to and from the format that the Miranda server understands, you're using Camel's Java 8 DSL to perform the message transformations ❸ ❹ in as few lines of code as possible.

You can find the code for this example in the `chapter9/miranda` folder of the book's source code, and you can try the example by using the following Maven goal:

```
mvn test -Dtest=MirandaTest  
mvn test -Dtest=MirandaJava8Test
```

You've now learned all about the Camel Test Kit and how to use it for unit testing with Camel. You looked at using the mock component to easily write tests with expectations, run tests, and have Camel verify whether the expectations were satisfied. You also saw how to use the mock component to simulate a real component. You may wonder whether there's a more cunning way to simulate a real component than by using a mock, and there is. You'll look at simulating errors next, but the techniques

involved could also be applied to simulating a real component.

9.4 Simulating errors

This section covers how to test that your code works when errors happen.

If your servers are on the premises and within a certain vicinity, you could test for errors by unplugging network cables and swinging an axe at the servers, but that's a bit extreme. If your servers are hosted in the cloud, you'll have a long walk and have to go all Chuck Norris to enter the guarded data centers of Amazon, Google, or whoever your cloud providers are. That's sadly not going to happen in our lifetime; we can only dream about becoming Neo in *The Matrix*, capable of learning ninja and kung-fu skills in a matter of minutes.

Back to our earthly lives—you have to come up with something you can do from your computer. You'll look at how to simulate errors in unit tests by using the three techniques listed in table 9.7.

Table 9.7 Three techniques for simulating errors

Technique	Description
Processor	Using processors is easy, and they give you full control as a developer. This technique is covered in section 9.4.1.
Mock	Using mocks is a good overall solution. Mocks are fairly easy to apply, and they provide a wealth of other features for testing, as you saw in section 9.3. This technique is covered in section 9.4.2.

In
te
rc
t
e
pt
o
r

This is the most sophisticated technique because it allows you to use an existing route without modifying it. Interceptors aren't tied solely to testing; they can be used anywhere and anytime. This technique is covered in section 9.4.3.

9.4.1 SIMULATING ERRORS USING A PROCESSOR

Errors are simulated in Camel by throwing exceptions, which is exactly how errors occur in real life. For example, Java will throw an exception if it can't connect to a remote server. Throwing such an exception is easy—you can do that from any Java code, such as from a Processor. That's the topic of this section.

To illustrate this, we'll take the use case from Rider Auto Parts: you're uploading reports to a remote server using HTTP, and you're using FTP as a fallback method. This allows you to simulate errors with HTTP connectivity.

The route from listing 11.14 is repeated here:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(5).redeliveryDelay(10000));

onException(IOException.class).maximumRedeliveries(3)
    .handled(true)
    .to("ftp://gear@ftp.rider.com?password=secret");

from("file:/rider/files/upload?delay=15m")
    .to("http://rider.com?user=gear&password=secret");
```

What you want to do now is simulate an error when sending a file to the HTTP service, and you'll expect that it'll be handled by `onException` and uploaded using FTP instead. This will ensure that the route is working correctly.

Because you want to concentrate the unit test on the error-handling aspect and not on the components used, you can mock the HTTP, FTP, and File endpoints. This frees you from the burden of setting up HTTP and FTP servers and leaves you with

a simpler route for testing:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(5).redeliveryDelay(1000));

onException(IOException.class).maximumRedeliveries(3)
    .handled(true)
    .to("mock:ftp");

from("direct:file")
    .to("mock:http");
```

This route also reduces the redelivery delay from 10 seconds to 1 second, to speed up unit testing. Notice that the file endpoint is replaced with the direct endpoint that allows you to start the test by sending a message to the direct endpoint; this is much easier than writing a file.

To simulate a communication error when trying to send the file to the HTTP endpoint, you add a processor to the route that forces an error by throwing a `ConnectException` exception:

```
from("direct:file")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception
{
            throw new ConnectException("Simulated error");
        }
    })
    .to("mock:http");
```

And with Java 8 DSL, it can be a bit shorter:

```
from("direct:file")
    .process(e -> { throw new ConnectException("Simulated
error"); })
    .to("mock:http");
```

You then write a test method to simulate this connection error, as follows:

```
@Test
public void testSimulateConnectionError() throws Exception
{
    getMockEndpoint("mock:http").expectedMessageCount(0);
```

```
    getMockEndpoint("mock:ftp").expectedBodiesReceived("Camel
rocks");

    template.sendBody("direct:file", "Camel rocks");

    assertMockEndpointsIsSatisfied();
}
```

You expect no messages to arrive at the HTTP endpoint because you predicted the error would be handled and the message would be routed to the FTP endpoint instead.

The book's source code contains this example. You can try it by running the following Maven goal from the chapter9/error directory:

```
mvn test -Dtest=SimulateErrorUsingProcessorTest
mvn test -Dtest=SimulateErrorUsingProcessorJava8Test
```

Using the Processor is easy, but you have to alter the route to insert the Processor. When testing your routes, you might prefer to test them *as is* without changes that could introduce unnecessary risks. What if you could test the route without changing it at all? The next two techniques do this.

9.4.2 SIMULATING ERRORS USING MOCKS

You saw in section 9.3.5 that the mock component could be used to simulate a real component. But instead of simulating a real component, you can use what you learned there to simulate errors. If you use mocks, you don't need to alter the route; you write the code to simulate the error directly into the test method, instead of mixing it in with the route. The following listing shows this.

[Listing 9.18](#) Simulating an error by throwing an exception from the mock endpoint

```
@Test
public void testSimulateConnectionErrorUsingMock() throws
Exception {
    getMockEndpoint("mock:ftp").expectedMessageCount(1);
```

```

MockEndpoint http = getMockEndpoint("mock:http");
http.whenAnyExchangeReceived(new Processor() {
    public void process(Exchange exchange) throws Exception
{
    throw new ConnectException("Simulated connection
error");
}
});

template.sendBody("direct:file", "Camel rocks");

assertMockEndpointsSatisfied();
}

```

To simulate the connection error, you need to get hold of the HTTP mock endpoint, where you use the `whenAnyExchangeReceived` method to set a custom Processor. That Processor can simulate the error by throwing the connection exception.

By using mocks, you put the code that simulates the error into the unit-test method, instead of in the route, as is required by the processor technique.

The source code contains an example in the `chapter9/error` directory that you can try by using the following Maven goal:

```
mvn test -Dtest=SimulateErrorUsingMockTest
```

Now let's look at the last technique for simulating errors.

9.4.3 SIMULATING ERRORS USING INTERCEPTORS

Suppose your boss wants you to write integration tests for the example from section 9.4.1 that should, among other things, test what happens when communication with the remote HTTP server fails. How can you do that? It's tricky because you don't have control over the remote HTTP server, and you can't easily force communication errors in the network layer. Luckily, Camel provides features to address this problem. We'll get to that in a moment, but first you need to look at interceptors, which provide

the means to simulate errors.

In a nutshell, an *interceptor* allows you to intercept any given message and act on it. [Figure 9.5](#) illustrates where the interception takes place in a route.

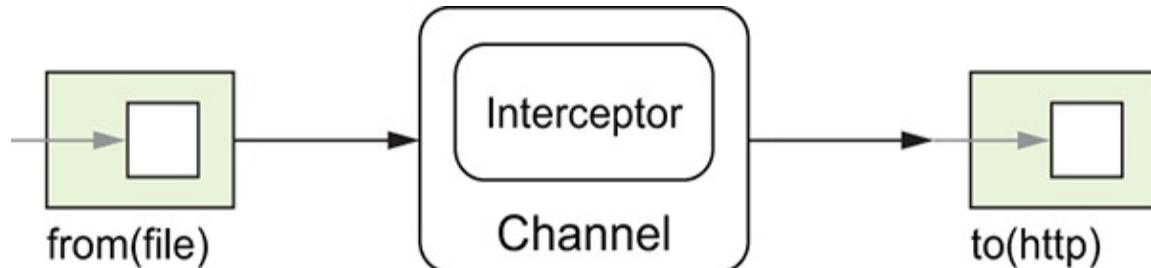


Figure 9.5 The channel acts as a controller, and it's where messages are intercepted during routing.

[Figure 9.5](#) shows a low-level view of a Camel route, whereby you route messages from a file consumer to an HTTP producer. In between sits the channel, which acts as a controller, and this is where the interceptors (among others) live.

Channels play a key role

Channels play a key role in the internal Camel routing engine, handling such things as routing the message to the next designated target, error handling, interception, tracing messages, and gathering metrics.

[Table 9.8](#) lists three types of interceptors that Camel provides out of the box.

Table 9.8 The three flavors of interceptors provided out of the box in Camel

Interceptor	Description
intercept	Intercepts every single step a message takes. This interceptor is invoked continuously as the message is routed.
interceptFromEndpoint	Intercepts incoming messages arriving at a particular endpoint. This interceptor is invoked only once.

interceptSendToEndpoint	Intercepts messages that are about to be sent to a particular endpoint. This interceptor is invoked only once.
-------------------------	--

To write integration tests, you can use `interceptSendToEndpoint` to intercept messages sent to the remote HTTP server and redirect them to a processor that simulates the error, as shown here:

```
interceptSendToEndpoint("http://rider.com/rider")
    .skipSendToOriginalEndpoint()
    .process(new SimulateHttpErrorProcessor());
```

When a message is about to be sent to the HTTP endpoint, it's intercepted by Camel, and the message is routed to your custom processor, where you simulate an error. When this detour is complete, the message would normally be sent to the originally intended endpoint, but you instruct Camel to skip this step using the `skipSendToOriginalEndpoint` method.

TIP The last two interceptors in table 9.8 support using wildcards (*) and regular expressions in the endpoint URL. You can use these techniques to intercept multiple endpoints or to be lazy and just match all HTTP endpoints. We'll look at this in a moment.

Because you're doing an integration test, you want to keep the original route *untouched*, which means you can't add interceptors or mocks directly in the route. Because you still want to use interceptors in the route, you need another way to somehow add the interceptors. Camel provides the `adviceWith` method to address this.

9.4.4 USING ADVICEWITH TO ADD INTERCEPTORS TO AN EXISTING ROUTE

The `adviceWith` method, available during unit testing, allows you to add such things as interceptors and error handling to an existing route.

To see how this works, let's look at an example. The following code snippet shows how to use `adviceWith` in a unit-test method:

```
@Test  
public void testSimulateErrorUsingInterceptors() throws  
Exception {  
    RouteDefinition route =  
context.getRouteDefinitions().get(0); ①
```

①

Selects the first route to be advised

```
route.adviceWith(context, new RouteBuilder() { ②
```

②

Uses `adviceWith` to add interceptor to route

```
public void configure() throws Exception {  
    interceptSendToEndpoint("http:///*")  
        .skipSendToOriginalEndpoint()  
        .process(new SimulateHttpErrorProcessor());  
    }  
});  
..  
}
```

The key issue when using `adviceWith` is to know which route to use. Because you have only one route in this case, you can refer to the first route enlisted in the route definitions list ①. The route definitions list contains the definitions of all routes registered in the current `CamelContext`.

When you have the route, it's only a matter of using the `adviceWith` method ②, which uses `RouteBuilder`. In the `configure` method, you can use the Java DSL to define the interceptors. Notice that the interceptor uses a wildcard to match all HTTP endpoints.

TIP If you have multiple routes, you'll need to select the correct route to be used. To help select the route, you can assign

unique IDs to the routes, which you then can use to look up the route, such as `context.getRouteDefinition("myCoolRoute")`.

We've included this integration test in the book's source code, in the chapter9/error directory. You can try the test by using the following Maven goal:

```
mvn test -Dtest=SimulateErrorUsingInterceptorTest
```

TIP Interceptors aren't only for simulating errors; they're a general-purpose feature that can also be used for other types of testing. For example, when you're testing production routes, you can use interceptors to detour messages to mock endpoints.

This example only scratches the surface of `adviceWith`. Let's take a moment to cover what else you can do.

9.4.5 USING ADVICEWITH TO MANIPULATE ROUTES FOR TESTING

`adviceWith` is used for testing Camel routes that are *advised* before testing. Here are some of the things you can do by advising:

- Intercept sending to endpoints
- Replace incoming endpoint with another
- Take out or replace node(s) of the route
- Insert new node(s) into the route
- Mock endpoints

All these features are available from `AdviceWithRouteBuilder`, which is a specialized `RouteBuilder`. This builder provides the additional fluent builder methods listed in table 9.9.

Table 9.9 Additional `adviceWith` methods from `AdviceWithRouteBuilder`

Method	Description
<code>mockEndpoints</code>	Mocks all endpoints in the route.
<code>mockEndpoints(patterns...)</code>	Mocks all endpoints in the route that matches the pattern(s). You can use wildcards and regular expressions in the given pattern to match multiple endpoints.
<code>mockEndpointsAndSkip(patterns...)</code>	Mocks all endpoints and skips sending to the endpoint in the route that matches the pattern(s). You can use wildcards and regular expressions in the given pattern to match multiple endpoints.
<code>replaceFromWith(uri)</code>	Easily replaces the incoming endpoint in the route.
<code>weaveById(pattern)</code>	Manipulates the route at the node IDs that matches the pattern.
<code>weaveByToString(pattern)</code>	Manipulates the route at the node string representation (output from <code>toString</code> method) that matches the pattern.
<code>weaveByToUri(pattern)</code>	Manipulates the route at the nodes sending to endpoints matching the pattern.
<code>weaveByType(pattern)</code>	Manipulates the route at the node type that matches the pattern.
<code>weaveAddFirst</code>	Easily weaves in new nodes at the beginning of the route.
<code>weaveAddLast</code>	Easily weaves in new nodes at the end of the route.

The rest of this section covers the first four methods in table 9.9, and all the `weave` methods are covered in section 9.4.6.

TELL CAMEL THAT YOU'RE ADVISING ROUTES

Before you jump into the pool, you need to learn a rule from the lifeguard. When `adviceWith` is being used, Camel will restart the advised routes. This occurs because `adviceWith` manipulates the route tree, and therefore Camel has to do the following:

- Stop the existing route
- Remove the existing route
- Add the route as a result of the `adviceWith`
- Start the new route

This happens for each of the advised routes during startup of your unit tests. It happens quickly: a route is started and then is immediately stopped and removed. This is an undesired behavior; you want to avoid restarts of the advised routes.

To solve this dilemma, you follow these three steps:

1. Tell Camel that the routes are being advised, which will prevent Camel from starting the routes during startup.
2. Advise the routes in your unit-test methods.
3. Start Camel after you've advised the routes.

The first step is done by overriding the `isUseAdviceWith` method from `CamelTestSupport` to return `true`:

```
public class MyAdviceWithTest extends CamelTestSupport {
```

```
    @Override
    public boolean isUseAdviceWith() {
```

Tells Camel that you intend to use `adviceWith` in this unit test

```
        return true;
    }
    ...
}
```

Then you advise the routes followed by starting Camel:

```
@Test
public void testMockEndpoints() throws Exception {
    RouteDefinition route =
context.getRouteDefinition("quotes");
    route.adviceWith(context, new AdviceWithRouteBuilder()
{
```

1

Advises the route

```
public void configure() throws Exception {  
    mockEndpoints(); ②
```

Automocks all the endpoints

```
}  
});  
context.start(); ③
```

Starts Camel after the advice is done

...

To advise a route, you need to obtain the route by using its route ID. Then you can use the `AdviceWithRouteBuilder` ① that allows you to use the advice methods from table 9.9, such as automocking all the Camel endpoints. After you're finished with the advice, you need to start Camel, which would then start the routes after they've been advised, and thus no restart of the routes is performed anymore.

You can try this example, provided as part of the source code in the `chapter9/advicewith` directory, by using the following Maven goal:

```
mvn test -Dtest=AdviceWithMockEndpointsTest
```

What was the important message from the lifeguard? To tell Camel that you're using `adviceWith`.

REPLACE ROUTE ENDPOINTS

You may have built Camel routes that start from endpoints that consume from databases, message brokers, cloud systems, embedded devices, social streams, or other external systems. To

make unit-testing these kind of routes easier, you can replace the route input endpoints with internal endpoints such as direct or SEDA endpoints. The following listing illustrates how to do this.

Listing 9.19 Replacing route endpoint from AWS SQS to SEDA for easy unit-testing

```
public class ReplaceFromTest extends CamelTestSupport {  
  
    @Override  
    public boolean isUseAdviceWith() {  
        return true;      ①  
  
    }  
  
    @Test  
    public void testReplaceFromWithEndpoints() throws  
Exception {  
    RouteDefinition route =  
context.getRouteDefinition("quotes");      ②  
  
    route.adviceWith(context, new AdviceWithRouteBuilder()  
{  
        public void configure() throws Exception {  
            replaceFromWith("direct:hitme");      ③  
  
            mockEndpoints("seda:*");      ④  
        }  
    }  
}
```

①

This test uses adviceWith.

}

@Test

```
    public void testReplaceFromWithEndpoints() throws  
Exception {  
    RouteDefinition route =  
context.getRouteDefinition("quotes");      ②
```

②

Gets the route to be advised

```
    route.adviceWith(context, new AdviceWithRouteBuilder())
```

{

```
        public void configure() throws Exception {  
            replaceFromWith("direct:hitme");      ③
```

③

Replaces the route input endpoint with a direct endpoint

```
        mockEndpoints("seda:*");      ④
```

④

Mocks all SEDA endpoints

```
    }  
});  
  
context.start(); 5
```

5

Starts Camel after you've finished advising

```
getMockEndpoint("mock:seda:camel") 6
```

6

Sets expectations on the mock endpoints

```
.expectedBodiesReceived("Camel rocks"); 6  
getMockEndpoint("mock:seda:other") 6  
.expectedBodiesReceived("Bad donkey"); 6  
  
template.sendBody("direct:hitme", "Camel rocks"); 7
```

7

Sends in test messages to the direct endpoint

```
template.sendBody("direct:hitme", "Bad donkey");  
  
assertMockEndpointsSatisfied(); 8
```

8

Asserts the test passed by asserting the mocks

```
}
```

```
@Override  
protected RoutesBuilder createRouteBuilder() throws  
Exception {  
    return new RouteBuilder() {  
        public void configure() throws Exception {  
            from("aws-sqs:quotes").routeId("quotes") 9
```

9

The route originally consumes from Amazon SQS queue system.

```
        .choice()
            .when(simple("${body} contains
'Camel'")).to("seda:camel")
            .otherwise().to("seda:other");
    }
}
}
```

The ReplaceFromTest class is using `adviceWith`, so you need to override the `isUseAdviceWithMethod` and return true **1**. Then you select the route to be advised **2** by using

`AdviceWithRouteBuilder`. The original input route is using the AWS-SQS component **9**, which you replace to use a direct component instead **3**. The route will send messages to other endpoints such as SEDA. To make testing easier, you automock all SEDA endpoints **4**. After the advice is done, you need to start Camel **5**. Because all the SEDA endpoints were automocked, you can use `mock:` as a prefix to obtain the mocked endpoints. This allows you to set expectations on those mock endpoints **6**. Then a couple of test messages are sent into the route by using the replaced endpoint, `direct:hitme` **7**. Finally, you check whether the test passes by asserting the mocks **8**.

You can try this example with the source code from the chapter9/adviceWith directory using the following Maven goal:

```
mvn test -Dtest=ReplaceFromTest
```

Replacing the input endpoint in Camel routes by using `adviceWith` is just the beginning of the route manipulation capabilities available. The following section covers how to rip and tear your routes as you please.

9.4.6 USING WEAVE WITH ADVICEWITH TO AMEND ROUTES

When testing your Camel routes, you can use `adviceWith` to *weave* the routes before testing.

Naming in IT

Some IT professionals say that naming is hard, and they were right in terms of `adviceWith` and `weave`. The name *amend* might have been a better choice than `weave`, because what you're doing is amending the routes before testing. *Weaving* allows you to remove or replace parts of the routes that may not be relevant for a particular unit test that's testing other aspects of the routes. Another use case is to replace parts of routes that couple to systems that aren't available for testing or may otherwise slow testing.

In this section, you'll try some of the `weave` methods from `AdviceWithRouteBuilder` listed in [table 9.9](#).

We'll start with a simple route that calls a bean that performs a message transformation:

```
from("seda:quotes").routeId("quotes")
    .bean("transformer").id("transform")
    .to("seda:lower");
```

Now suppose invoking the bean requires a connection to an external system that you don't want to perform in a unit test. You can use `weaveById` to select the bean and then replace it with another kind of transformation:

```
public void testWeaveById() throws Exception {
    RouteDefinition route =
        context.getRouteDefinition("quotes");
    route.adviceWith(context, new AdviceWithRouteBuilder() {
        public void configure() throws Exception {
            weaveById("transform").replace() ①
                .transform().simple("${body.toUpperCase()}"); ②
        }
    });
}
```

①

Uses `weave` to select the node to be replaced

②

```
weaveAddLast().to("mock:result"); ②
```

②

Routes to a mock endpoint at the end of the route

```
}
```

```
});  
context.start(); ③
```

③

Starts Camel after adviceWith

```
getMockEndpoint("mock:result").expectedBodiesReceived("HELL  
O CAMEL"); ④
```

④

Expects the message to be in uppercase

```
template.sendBody("seda:quotes", "Hello Camel");  
assertMockEndpointsSatisfied();  
}
```

In the original route, the bean call was given the ID `transform`, which is the ID you let `weaveById` ① use as select criteria. Then you select an *action* that can be one of the following:

- `remove`—Removes the selected nodes.
- `replace`—Replaces the selected nodes with the following.
- `before`—Before each of the selected nodes, the following is added.
- `after`—After each of the selected nodes, the following is added.

In this example, you want to replace the node, so you use `replace`, which is using the Java DSL to define a mini Camel route that's the replacement. In this example, you replace the bean with a `transform`.

You aren't restricted to only a 1:1 replacement, so you could, for example, provide before and after logging as shown here:

```
weaveById("transform").replace()
    .log("Before transformation")
    .transform().simple("${body.toUpperCase()}")
    .log("Transformation done");
```

To make testing easier, you use `weaveAddLast` ❷ to route to a mock endpoint at the end of the route. Remember to start Camel ❸ before you start sending messages into the route and assert that the test is correct ❹.

You can play with this example from the source code in `chapter9/adviceWith` by running this Maven goal:

```
mvn test -Dtest=WeaveByIdTest
```

Using `weaveById` is recommended if you've assigned IDs to the EIPs in your Camel routes. If this isn't the case, what can you do then?

WEAVE WITHOUT USING IDS

When weaving a route, you need to use one of the `weaveBy` methods listed in table 9.9 as criteria to select one or more nodes in the route graph. Suppose you use the Splitter EIP in a route; then you can use `weaveByType` to select this EIP. We've prepared a little example for you that uses the Splitter EIP in the route shown here:

```
from("seda:quotes").routeId("quotes")
    .split(body())
flexible().accumulateInCollection(ArrayList.class)
    .transform(simple("${body.toLowerCase()}"))
    .to("mock:line")
.end()
.to("mock:combined");
```

TIP The aggregation strategy provided to the Splitter EIP uses a flexible fluent builder (highlighted in bold) that allows you to

build common strategies. You can use this builder by statically importing the org.apache.camel.util.toolbox.AggregationStrategies.flexible method.

Because the route has only one Splitter EIP, you can use weaveByType to find this single splitter in the route. Using weaveByType requires you to pass in the model type of the EIP. The name of the model type uses the naming pattern **nameDefinition**, so the splitter is named **SplitDefinition**:

```
weaveByType(SplitDefinition.class)
    .before()
        .filter(body().contains("Donkey"))
        .transform(simple("${body}, Mules cannot do this"));
```

Here you weave and select the Splitter EIP and then insert the following route snippet before the splitter. The route snippet is a message filter that uses a predicate to check whether the message body contains the word "Donkey". If so, it performs a message transformation by appending a quote to the existing message body.

As usual, we've prepared this example of wisdom you can try from the chapter9/advice directory using the following Maven goal:

```
mvn test -Dtest=WeaveByTypeTest
```

Okay, this example has only one splitter. What if you have more than one splitter or want to weave a frequently used node such as to?

SELECTING ONE OR MORE BY USING WEAVE

When using the weaveBy methods, they select all matching nodes, which can be anything from none, one, two, or more nodes. In those situations, you may want to narrow the selection to a specific node. This can be done by using the select methods:

- `selectFirst`—Selects only the first node.
- `selectLast`—Selects only the last node.
- `selectIndex(index)`—Selects only the n th node. The index is zero based.
- `selectRange(from, to)`—Selects the nodes within the given range. The index is zero based.
- `maxDeep(level)`—Limits the selection to at most N levels deep in the Camel route tree. The first level is number 1. So number 2 is the children of the first-level nodes.

Given the following route, you want to select the highlighted `to` node:

```
from("seda:quotes").routeId("quotes")
    .split(body(),
flexible().accumulateInCollection(ArrayList.class))
    .transform(simple("${body.toLowerCase()}"))
    .to("seda:line")
    .end()
    .to("mock:combined");
```

You can then use `weaveByType` and `selectFirst` to match only the first found:

```
weaveByType(ToDefinition.class).selectFirst().replace().to(
    "mock:line");
```

You can try this example from the accompanying source code in the chapter9/advice directory using the following Maven goal:

```
mvn test -Dtest=WeaveByTypeSelectFirstTest
```

If the route uses more `sent to` EIPs, the select criteria changes. You would then need to use `selectByIndex` and count the number of `to` occurrences from top to bottom, to find the index to use.

This last example can be made even easier by using `weaveByToUri`, which matches sending to endpoints:

```
weaveByToUri("seda:line").replace().to("mock:line");
```

You can try this yourself with the source code from the chapter9/adviceWith directory by running the following Maven goal:

```
mvn test -Dtest=WeaveByToUriTest
```

Selecting nodes by using patterns

Both `weaveByToString` and `weaveByToUri` use pattern matching to match the nodes to select. The pattern algorithm is based on the same logic used by Camel interceptors. For example, to match all endpoints that send to a SEDA endpoint, you can use "`seda:*`". Regular expressions can be used, such as "`.*gold.*`", to match any nodes that have the word `gold` in the endpoint URI.

That's all, folks, that we wanted to talk about `adviceWith`. It's a powerful feature in the right hands, so we recommend you give it a shot and play with the examples in the source code.

The two last sections of this chapter cover Camel integration testing. Section 9.5 focuses on integration testing using the Camel Test Kit. Section 9.6 goes out of town to look at using three third-party test frameworks with Camel.

Do you need a break? If so, now is a good time to put on the kettle for a cup of tea or coffee. The last two sections are best enjoyed having a cup of hot brew readily available.

9.5 Camel integration testing

So far in this chapter, you've learned that mocks play a central role when testing Camel applications. But integration testing often involves real live components, and substituting them with mocks isn't an option, because the point of the integration test is

to test with live components. In this section, you’ll look at how to test such situations without mocks.

Rider Auto Parts has a client application that business partners can use to submit orders. The client dispatches orders over JMS to an incoming order queue at the Rider Auto Parts message broker. A Camel application is then used to further process these incoming orders. [Figure 9.6](#) illustrates this.

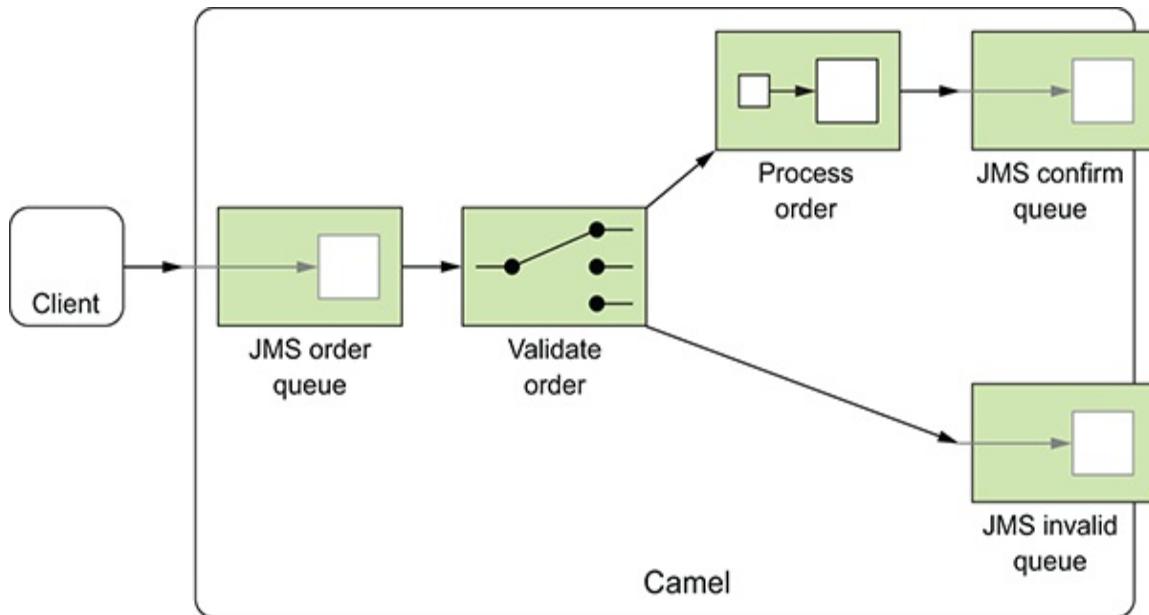


Figure 9.6 The client sends orders to an order queue, which is routed by a Camel application. The order is either accepted and routed to a confirm queue, or it's not accepted and is routed to an invalid queue.

The client application is written in Java, but it doesn’t use Camel at all. The challenge you’re facing is how to test that the client and the Camel application work as expected. How can you do integration testing?

9.5.1 PERFORMING INTEGRATION TESTING

Integration-testing the scenario outlined in [figure 9.6](#) requires you to use live components. You must start the test by using the client to send a message to the order queue. Then you let the Camel application process the message. When this is complete, you’ll have to inspect whether the message ended up in the right queue—the confirm or the invalid queue.

You have to perform these three tasks:

1. Use the client to send an order message.
2. Wait for the Camel application to process the message.
3. Inspect the confirm and invalid queues to see whether the message arrived as expected.

Let's tackle each step.

USE THE CLIENT TO SEND AN ORDER MESSAGE

The client is easy to use. All you're required to do is provide an IP address to the remote Rider Auto Parts message broker and then use its `sendOrder` method to send the order.

The following code has been simplified in terms of the information required for order details:

```
OrderClient client = new OrderClient("localhost:61616");
client.sendOrder(123, date, "4444", "5555");
```

WAIT FOR THE CAMEL APPLICATION TO PROCESS THE MESSAGE

The client has sent an order to the order queue on the message broker. The Camel application will now react and process the message according to the route outlined in [figure 9.6](#).

The problem you're facing now is that the client doesn't provide any API you can use to wait until the process is complete. You need an API that provides insight into the Camel application. All you need to know is when the message has been processed, and optionally whether it completed successfully or failed.

Camel provides `NotifyBuilder`, which provides such insight. Section 9.5.2 covers `NotifyBuilder` in more detail, but the following code shows how `NotifyBuilder` notifies you when Camel is finished processing the message:

```
NotifyBuilder notify = new
NotifyBuilder(context).whenDone(1).create();
```

```
OrderClient client = new  
OrderClient("tcp://localhost:61616");  
client.sendOrder(123, date, "4444", "5555");  
  
boolean matches = notify.matches(5, TimeUnit.SECONDS);  
assertTrue(matches);
```

First, you configure `NotifyBuilder` to notify you when one message is done. Then you use the client to send the message. Invoking the `matches` method on the `notify` instance will cause the test to wait until the condition applies or until the five-second time-out occurs.

The last task tests whether `NotifyBuilder` matches. If it didn't match, it's because the condition was not met within the time-out period, and the assertion would fail by throwing an exception.

INSPECT THE QUEUES TO SEE IF THE MESSAGE ARRIVED AS EXPECTED

After the message has been processed, you need to investigate whether the message arrived at the correct message queue. If you want to test that a valid order arrived in the confirm queue, you can use the `BrowsableEndpoint` to browse the messages on the JMS queue. Using `BrowsableEndpoint`, you only peek inside the message queue, which means the messages will still be present on the queue.

Doing this requires a little bit of code, as shown here:

```
BrowsableEndpoint be =  
context.getEndpoint("activemq:queue:confirm",  
  
BrowsableEndpoint.class);  
List<Exchange> list = be.getExchanges();  
assertEquals(1, list.size());  
String body = list.get(0).getIn().getBody(String.class);  
assertEquals("OK,123,2017-04-20T15:47:58,4444,5555", body);
```

Using `BrowsableEndpoint`, you can retrieve the exchanges on the JMS queue using the `getExchanges` method. You can then use the exchanges to assert that the message arrived as expected.

The source code for the book contains this example in the chapter9/notify directory; you can try the example by using the following Maven goal:

```
mvn test -Dtest=OrderTest
```

We've now covered an example of performing integration testing without mocks. Along the road, we introduced `NotifyBuilder`, which has many more nifty features. We'll review those in the next section.

9.5.2 USING NOTIFYBUILDER

`NotifyBuilder` is located in the `org.apache.camel.builder` package. It uses the Builder pattern, which means you stack methods on it to build an expression. You use it to define conditions for messages being routed in Camel. Then it offers methods to test whether the conditions have been met. We already used it in the previous section, but this time we'll raise the bar and show you how to build more complex conditions.

In the previous example, you used a simple condition:

```
NotifyBuilder notify = new  
NotifyBuilder(context).whenDone(1).create();
```

This condition will match when one or more messages have been processed in the entire Camel application. This is a coarse-grained condition. Suppose you have multiple routes, and another message is processed as well. That would cause the condition to match even if the message you wanted to test is still in progress.

To remedy this, you can pinpoint the condition so it applies only to messages originating from a specific endpoint, as shown in bold:

```
NotifyBuilder notify = new NotifyBuilder(context)  
.from("activemq:queue:order").whenDone(1).create();
```

Now you've told the notifier that the condition applies only to messages that originate from the order queue.

Suppose you send multiple messages to the order queue, and you want to test whether a specific message was processed. You can do that using a predicate to indicate when the desired message was completed, using the `whenAnyDoneMatches` method, as shown here in bold:

```
NotifyBuilder notify = new NotifyBuilder(context)
    .from("activemq:queue:order")
    .whenAnyDoneMatches(
        body().isEqualTo("OK,123,2017-04-
20'T'15:48:00,2222,3333"))
    .create();
```

In this example, you want the predicate to determine whether the body is equal to the expected result, which is the string starting with `OK, 123, ...`.

We've now covered some examples using `NotifyBuilder`, but the builder has many methods that allow you to build even more complex expressions. [Table 9.10](#) lists the most commonly used methods.

Table 9.10 Noteworthy methods on NotifyBuilder

Method	Description
<code>from(uri)</code>	Specifies that the message must originate from the given endpoint. You can use wildcards and regular expressions in the given URI to match multiple endpoints. For example, you could use <code>from("activemq:queue:*)</code> to match any ActiveMQ queues.
<code>fromRoute(routeId)</code>	Specifies that the message must originate from the given route. You can use wildcards and regular expressions in the given <code>routeId</code> to match multiple routes.
<code>filter(predicate)</code>	Specifies the the message must match the given predicate.
<code>wereSentTo(uri)</code>	Matches when a message at any point during the route was sent to the endpoint with the given URI. You can use wildcards and regular expressions in the given URI to match multiple endpoints.
<code>whenDone(number)</code>	Matches when a minimum number of messages are done.

<code>whenCompleted(number)</code>	Matches when a minimum number of messages are completed.
<code>whenFailed(number)</code>	Matches when a minimum number of messages have failed.
<code>whenBodiesDone(bodies...)</code>	Matches when messages are done with the specified bodies in the given order.
<code>whenAnyDoneMatches(predicate)</code>	Matches when any message is done and matches the predicate.
<code>create</code>	Creates the notifier.
<code>matches</code>	Tests whether the notifier currently matches. This operation returns immediately.
<code>matches(timeout)</code>	Waits until the notifier matches or times out. Returns <code>true</code> if it matches, or <code>false</code> if a time-out occurs.

NotifyBuilder has more than 30 methods; we've listed only the most commonly used ones in table 9.10. Consult the online Camel documentation to see all the supported methods: <http://camel.apache.org/notifybuilder.html>.

NOTE NotifyBuilder works in principle by adding EventNotifier to the given CamelContext. EventNotifier then invokes callbacks during the routing of exchanges. This allows NotifyBuilder to listen for those events and react accordingly. EventNotifier is covered in chapter 16.

NotifyBuilder identifies three ways a message can complete:

- *Done*—The message is done, regardless of whether it completes

or fails.

- *Completed*—The message completes with success (no failure).
- *Failed*—The message fails (for example, an exception is thrown and not handled).

The names of these three ways are also incorporated in the names of the builder methods: `whenDone`, `whenCompleted`, and `whenFailed` (listed in table 9.10).

TIP You can create multiple instances of `NotifyBuilder` if you want to be notified of different conditions. `NotifyBuilder` also supports using binary operations (and, or, not) to stack together multiple conditions.

The book’s source code contains examples of using `NotifyBuilder` in the `chapter9/notify` directory. You can run them by using the following Maven goal:

```
mvn test -Dtest=NotifyTest
```

We encourage you to take a look at this source code and the online documentation.

So far, we’ve covered testing Camel by using its own set of test modules, a.k.a. the Camel Test Kit. But we don’t live in a world of only Camels (although the authors of this book may have Camel too much on their minds). Numerous great test libraries are available for you to use. We’ll end this chapter by presenting some of these test libraries you can use with Camel.

9.6 Using third-party testing libraries

Over the years, several great testing frameworks have emerged. They’re worth introducing to you, and we provide basic examples of using them to test your Camel applications. These test frameworks aren’t a direct replacement for the Camel Test Kit.

These frameworks are suited for integration and system testing, whereas the Camel Test Kit is particularly strong for unit testing.

The third-party testing libraries covered in this section are as follows:

- *Arquillian*—Helps test applications that run inside a Java EE container
- *Pax Exam*—Helps test applications that run inside an OSGi container
- *Rest Assured*—Java DSL for easy testing of REST services
- *Other testing libraries*—A brief list of other kinds of testing libraries

The first two are test frameworks for system and integration testing. Rest Assured is a great test library that makes sending and verifying REST-based applications using JSON/XML payloads easier. You'll use this library in the first subsection to follow. At the end of the section, we quickly cover other testing libraries that we want to bring to your attention.

We start from the top with Arquillian.

9.6.1 USING ARQUILLIAN TO TEST CAMEL APPLICATIONS

Arquillian was created to help test applications running inside a Java EE server. Readers who have built Java EE applications may know that testing your deployments and applications on said servers isn't an easy task. The bigger and older the Java EE server is, the worse that task is. It's not unusual to find yourself developing and testing locally, using a lighter Java EE server such as JBoss Application Server, and then crossing your fingers when testing on a production-like system.

The use of Java EE servers has peaked, and Arquillian has also evolved to become a testing framework that allows developers to create automated integration, functional, and acceptance tests. Arquillian can also be used for integration testing Docker

containers and much more.

TIP You can learn a lot more about Arquillian from *Testing Java Microservices* by Alex Soto Bueno and Jason Porter (Manning, 2018).

In this section, we'll show you how to build an integration test with Arquillian that tests a Camel application running as a web application on a Java EE server such as Jetty or WildFly.

THE CAMEL EXAMPLE

The Camel application is a simple REST service that runs as a web application. The meat of this application is a Camel RESTful service that's defined using XML DSL notation in the quote.xml file, as shown in the following listing.

[Listing 9.20](#) Camel RESTful service defined using REST DSL (quote.xml)

```
<bean id="quote" class="camelaction.Quote"/> ①
```

①

A bean that returns a quote

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

```
  <restConfiguration component="servlet"/> ②
```

②

Uses the servlet component as the RESTful HTTP server

```
    <rest produces="application/json"> ③
```

③

3

RESTful service that produces JSON as output

```
<get uri="/say"> 4
```

4

GET /say service

```
<to uri="bean:quote"/> 5
```

5

Calls the bean to get a quote to return as response

```
</get>  
</rest>  
  
</camelContext>
```

This code uses Camel's Rest DSL to define a RESTful service. This is a sneak peak of what's coming in the next chapter, where REST services are on the menu. The example uses a Java bean that provides a quote from a wise man ①. The REST service uses the servlet component ② to tap into the servlet container of the Java EE server. The REST service has one GET service ④ that calls the bean ⑤ and returns the response as JSON ③.

The quote bean is a simple class with a single method returning one of the three great wise sayings, as shown in the following listing.

Listing 9.21 A quote bean with great universal wisdom

```
public class Quotes {  
  
    public static String[] QUOTES = new String[]{"Camel  
rocks",  
                                              "Donkeys are bad",  
                                              "We like beer"};  
    public String say() {  
        int idx = new Random().nextInt(3);  
        return QUOTES[idx];  
    }  
}
```

```
        return String.format("{\"quote\": \"%s\"}",  
QUOTES[idx]);  
    }  
}
```

The final piece of the application is the web.xml file, shown next.

Listing 9.22 web.xml to set up Camel servlet and bootstrap the Camel XML-based application

```
<web-app>  
  
    <context-param>  
        <param-name>contextConfigLocation</param-name>  
        <param-value>classpath:camelaction/quote.xml</param-  
value>    ①
```

①

Location of the Camel XML DSL file

```
    </context-param>  
  
    <listener>  
        <listener-  
class>org.springframework.web.context.ContextLoaderListener  
    ②
```

②

Uses Spring to bootstrap the Camel XML application

```
        </listener-class>  
    </listener>  
  
    <servlet>  
        <servlet-name>CamelServlet</servlet-name>  
        <servlet-  
class>org.apache.camel.component.servlet.CamelHttpTransport  
Servlet</servlet-class>    ③
```

③

Sets up the Camel Servlet

```
<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>CamelServlet</servlet-name>
  <url-pattern>/*</url-pattern> ④
```

④

Uses the root path for the Camel Servlet mapping

```
</servlet-mapping>

</web-app>
```

This web.xml file is an old and trusty way of bootstrapping a web application. At first, you reference the classpath location of the Spring XML file ① that includes the Camel route. Then you use Spring listener ② to bootstrap your application. The Camel Servlet component is configured as a servlet ③ with a URL mapping ④.

We'll reuse this example or a modification thereof in the following sections, so we took a bit of time in this section to present the full code listing. Now it's time to pay attention, because Arquillian enters the stage.

SETTING UP ARQUILLIAN

When using Arquillian, you need to take three steps to create and use an Arquillian test:

1. Add Arquillian artifacts to Maven pom.xml.
2. Choose the application server to run the tests.
3. Write a test class with the deployment unit and the actual test.

ADDING THE ARQUILLIAN ARTIFACTS

When using Arquillian, it's recommended to import the Arquillian Maven BOM in your pom.xml file, which makes setting up the right Maven dependencies easier. This is done by

adding arquillian-bom and arquillian-junit-container as dependencies to the Maven pom.xml file.

CHOOSING THE APPLICATION SERVER

Because Arquillian runs the tests inside a Java EE or web container, you need to pick which one to use for testing. We used WildFly in section 9.2, so let's use a different container this time—we'll choose Jetty by adding arquillian-jetty-embedded-9 as a dependency to the Maven pom.xml file.

Adding the arquillian-jetty-embedded-9 dependency doesn't include Jetty itself, so you need to add jetty-webapp, jetty-deploy, and jetty-annotations as well.

The Jetty dependencies should be set with <scope>runtime</scope> because Jetty is used only as the runtime Java EE container—you don't want to have compile-time dependencies on Jetty. You want your Camel web application to be portable and to run on other Java EE containers.

You can see the exact details of these Maven dependencies from the source code located in the chapter9/arquillian-web directory.

You're now ready to write the integration test.

WRITING THE TEST CLASS WITH THE DEPLOYMENT UNIT AND ACTUAL TEST

Let's jump straight into code that shows the Arquillian test.

Listing 9.23 Arquillian test that creates a deployment unit and test method

```
@RunWith(Arquillian.class)
```

1

1

Runs the test with Arquillian

```
public class QuoteArquillianTest {
```

```
    @Deployment      ②
```

②

Sets up the deployment unit in this method

```
    public static Archive<?> createTestArchive() {  
        return ShrinkWrap.create(WebArchive.class)      ③
```

③

Uses ShrinkWrap to create a micro deployment unit

```
        .addClass(Quotes.class)      ③  
        .setWebXML(Paths.get("src/main/webapp/WEB-  
INF/web.xml").toFile());      ③  
    }
```

```
    @Test  
    @RunAsClient      ④
```

④

Runs this test as a client

```
    public void testQuote(@ArquillianResource URL url) throws  
Exception {      ⑤
```

⑤

Injects the URL of the deployed application

```
        given().      ⑥
```

⑥

Uses Rest Assured to verify REST call

```
            baseUri(url.toString()).      ⑥  
            when().      ⑥  
            get("/say").      ⑥  
            then().      ⑥  
            body("quote", isIn(Quotes.QUOTES));      ⑥
```

```
}
```

An Arquillian test is a JUnit test that's been annotated with `@RunWith(Arquillian.class)` ❶. The deployment unit is defined by a static method that's been annotated with `@Deployment` ❷. The method creates an Archive using ShrinkWrap; you're in full control of what to include in the deployment unit ❸. In this example, you need only the Quotes class and the web.xml file.

The big picture of ShrinkWrap

ShrinkWrap is used to build the test archive that's deployed into the application server. This allows you to build isolated test deployments that include only what you need. From the application server point of view, the ShrinkWrap archive is a real deployment unit.

The Camel application you want to test uses the Camel servlet component. Therefore, you want to start the test from the client by calling the servlet, and you need to annotate the test with `@RunAsClient` ❹. Otherwise, the test will be run inside the application server. The URL to the deployment application is injected into the test method by the `@ArquillianResource` annotation ❺.

The implementation of the test method uses the Rest Assured test library that provides a great DSL to define the REST-based test ❻. The code should reason well with Camel riders who are using the Camel DSL to define integration routes.

This example is provided with the source code in the `chapter9/arquillian-web` directory; you can try the example using the following Maven goal:

```
mvn test
```

There's a lot more to Arquillian than what we can cover in this

book. If you're running Camel applications in an application server, we recommend that you give Arquillian a try.

And speaking of application servers, let's take a look at a different kind: the OSGi-based Apache Karaf.

9.6.2 USING PAX EXAM TO TEST CAMEL APPLICATIONS

Pax Exam is for OSGi what Arquillian is for Java EE servers. If the mountain won't come to Muhammad, Muhammad must go to the mountain. This phrase covers the principle of both Arquillian and Pax Exam. Your tests must go to those application servers.

Earlier in this chapter, section 9.2.4 covered testing Camel by using OSGi Blueprint with the camel-test-blueprint module. This module has its limitations. In this section, you'll use Pax Exam to test the Camel quote application built in the previous section. Last time, the quote application was deployed as a web application in Jetty; this time, you'll use OSGi and deploy the application as an OSGi bundle.

MIGRATING THE CAMEL APPLICATION FROM WEB APPLICATION TO OSGI BUNDLE

You migrate the Camel application from a web application by doing the following:

1. Change the pom.xml from a WAR file to an OSGi bundle.
2. Add the Maven Bundle plugin to the pom.xml file.
3. Create a features.xml file to make installing the application easy.
4. Convert the Spring XML file to OSGi Blueprint XML.

You start by changing the pom.xml from a web application to OSGi by setting packaging to bundle:

```
<packaging>bundle</packaging>
```

Then in the plugins section, you add the Maven Bundle Plugin

that will generate the OSGi information in the MANIFEST.MF file, which turns the generated JAR into an OSGi bundle.

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>3.3.0</version>
  <extensions>true</extensions>
</plugin>
```

You then add another plugin that attaches the features.xml file to the project as a Maven artifact. This ensures that the features file gets installed and released to Maven, making it easy to install this example in Apache Karaf. The features.xml file defines which Camel and Karaf features this example requires and will be installed together with the example itself:

```
<features
  xmlns="http://karaf.apache.org/xmlns/features/v1.2.0"
    name="chapter9-pax-exam">
  <feature name="camel-quote" version="2.0.0">
    <feature>war</feature> 1
```

1

Installs the Karaf WAR feature, which includes HTTP servlet support

```
    <feature>camel-blueprint</feature> 2
```

2

Installs these Camel components

```
      <feature>camel-servlet</feature> 2
      <bundle>mvn:com.cameleinaction/chapter9-pax-
exam/2.0.0</bundle> 3
```

3

Installs the Camel example itself

```
    </feature>
</features>
```

The biggest change in the example is the migration from Spring XML to OSGi Blueprint. It's not the differences between Spring and Blueprint that result in the big changes, as the syntax is similar. It's the fact that you need to set up the Camel servlet component to use the OSGi HTTP service in order to tap into the servlet container from the Apache Karaf server. The following listing shows how this is done.

Listing 9.24 Camel OSGi Blueprint using servlet with OSGi HTTP service

```
<blueprint
  xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="
    http://www.osgi.org/xmlns/blueprint/v1.0.0

https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

<reference id="httpService"
  interface="org.osgi.service.http.HttpService"/> ①
```

①

Refers to use the OSGi HTTP Service

```
<bean
  class="org.apache.camel.component.servlet osgi.OsgiServletR
egisterer"
  init-method="register" ②
```

②

Sets up a Servlet that uses /camel as context-path

```
  destroy-method="unregister">
<property name="alias" value="/camel"/>
<property name="httpService" ref="httpService"/>
<property name="servlet" ref="quotesServlet"/>
</bean>
```

```
<bean id="quotesServlet" ③
```

③

Camel servlet

```
class="org.apache.camel.component.servlet.CamelHttpTransportServlet"/>
```

```
<bean id="quote" class="camelaction.Quote"/>
```

Bean with the quotes

```
<camelContext id="quotesCamel"  
xmlns="http://camel.apache.org/schema/blueprint">  
    <restConfiguration component="servlet"
```

Sets up a Servlet that uses /camel as context-path

```
        port="8181" contextPath="/camel"/>  
    <rest produces="application/json">
```

RESTful service that produces JSON as output

```
        <get uri="/say">
```

GET /say service

```
            <to uri="bean:quote"/>
```

Calls the bean to get a quote to return as response

```
        </get>  
    </rest>  
</camelContext>
```

```
</blueprint>
```

The Camel application uses the servlet container provided by Apache Karaf, by referring to the OSGi HTTP service ①. Then you use a bit of XML to set up a servlet that uses the context-path /camel ② and the Camel servlet to be used ③. The rest of the example is similar to the example used when testing with Arquillian.

RUNNING THE EXAMPLE

This example is included with the source code in chapter9/pax-exam, and you can build and run the example in Apache Karaf. First, you need to build the example from Maven using the following:

```
mvn install
```

Then you can start Apache Karaf. From the Karaf shell, type the following commands:

```
feature:repo-add camel 2.20.1
feature:install camel
feature:repo-add mvn:com.camerlaction/chapter9-pax-
exam/2.0.0/xml/features
feature:install camel-quote
```

And from a web browser, you can open the following URL:

```
http://localhost:8181/camel/say
```

The web browser should show a quote as the response, such as the following:

```
{
    quote: "Camel rocks"
}
```

Okay, now it's time to listen up and cover your hair in your eyes (hint: a little tribute to Nirvana performing their MTV unplugged album over two decades ago).

SETTING UP PAX EXAM

You need to take two steps to create and use a Pax Exam test:

1. Add Pax Exam artifacts to Maven pom.xml.
2. Write a test class with the deployment unit and the actual test.

ADDING PAX EXAM ARTIFACTS

Adding Pax Exam to Maven pom.xml requires adding the following dependencies: pax-exam, pax-exam-junit4, pax-exam-inject, pax-exam-link-mvn, and pax-exam-container-karaf.

You can find the full list of the Pax Exam Maven artifacts in the source code for this example, located in the chapter9/pax-exam directory.

WRITING THE INTEGRATION TEST CLASS

Let's jump straight into the code and follow up with an explanation of the important details.

Listing 9.25 Pax Exam Integration test class with deployment and actual test

```
@RunWith(PaxExam.class)
```

1

1

Runs this test with Pax Exam

```
public class PaxExamIT {  
  
    @Inject  
    @Filter("(camel.context.name=quotesCamel)")
```

2

2
Injects the CamelContext from the application

```
    protected CamelContext camelContext;
```

```
@Configuration ③
```

③

Sets up the deployment unit

```
public Option[] config() {  
    return new Option[]{  
        karafDistributionConfiguration() ④
```

④

Configures the Apache Karaf server to use

```
.frameworkUrl(maven().groupId("org.apache.karaf") ④  
    .artifactId("apache-  
karaf").version("4.1.2").type("tar.gz")) ④  
    .karafVersion("4.1.2") ④  
    .name("Apache Karaf") ④  
    .useDeployFolder(false) ④  
    .unpackDirectory(new File("target/karaf")), ④  
  
    keepRuntimeFolder(), ⑤
```

⑤

Keeps the runtime folder so we can look there in case of errors

```
configureConsole().ignoreRemoteShell(),  
logLevel(LogLevelOption.LogLevel.WARN), ⑥
```

⑥

Sets the logging level to not be verbose

```
junitBundles(),  
features(getCamelKaraffFeatureUrl(), "camel", "camel-  
test"), ⑦
```

⑦

Installs Apache Camel

```
    features(getCamelKarafFeatureUrl(), "camel-  
http4"), 8
```

8

Installs the camel-http4 component we use in this test

```
features(maven().groupId("com.camelinaction"))
```

9

9

Installs this example using the Maven coordinates

```
.artifactId("chapter9-paxexam").version("2.0.0")  
.classifier("features").type("xml"), "camel-quote")  
};  
  
}  
  
public static UrlReference getCamelKarafFeatureUrl() {  
    return mavenBundle()  
        .groupId("org.apache.camel.karaf").artifactId("apache-  
camel")  
  
.version("2.20.1").classifier("features").type("xml");  
}  
  
@Test  
public void testPaxExam() throws Exception { 10
```

10

The actual test method that runs inside Karaf container

```
long total =  
camelContext.getManagedCamelContext().getExchangesTotal();  
assertEquals("Should be 0 exchanges completed", 0,  
total);
```

```
Thread.sleep(2000); 11
```

11

Pax-Exam needs a little delay before really ready

```
String url = "http4://localhost:8181/camel/say";
```

```
ProducerTemplate template =  
camelContext.createProducerTemplate();  
String json = template.requestBody(url, null,  
String.class); ⑫
```

⑫

Calls the Camel Servlet and prints the response

```
System.out.println("Wiseman says: " + json);  
  
total =  
camelContext.getManagedCamelContext().getExchangesTotal();  
assertEquals("Should be 1 exchanges completed", 1,  
total);  
}
```

As you can see, you create a Pax Exam integration test by annotating the test class with `@RunWith(PaxExam.class)` **①**. You can perform dependency injection by using `@Inject`, which injects the `camelContext` of the Camel application being tested **②**. Then you configure what to deploy and run on the Apache Karaf server in the `@Configuration` method **③**. At first, you configure the Apache Karaf server **④**, which will be unpacked in the `target/karaf` directory and started from there. It's a good idea to keep the unpacked directory after the test **⑤**, which allows you to peek inside the Karaf server files, such as the log files. By default, Pax Exam is verbose in the logging, so you turn it up all the way to `WARN` level **⑥** to keep the logging to a minimum. But you can adjust this to `INFO` or `DEBUG` to have a lot more logging in case something is wrong. Then you install Apache Camel **⑦** and the `camel-http4` component **⑧**, which you use for testing your Camel application that's installed next **⑨**.

The remainder of the code is the test method **⑩**, which uses the injected `camelContext` **②** to create a `ProducerTemplate` that you use to call the Camel servlet **⑫**, after a little delay of two seconds **⑪**. The delay is needed because Pax Exam starts the test too quickly, while Karaf isn't yet ready to accept traffic on its HTTP server. Notice you assert that Camel has processed no message before the test, and then one message afterward. The

source code example includes an additional test to verify that the returned message is one of the expected quotes. But we left that code out of the listing.

RUNNING THE TEST

Now is a good time to run the Pax Exam integration test yourself from the chapter9/pax-exam directory, which is done via the following:

```
mvn integration-test -P pax
```

First-time users with Pax Exam will have a lot to learn and master, as setting up the `@Configuration` method can be tricky. Pax Exam has many options—more than what we can cover in this book. But we've given you a good start.

The last, brief section highlights numerous testing libraries. Each is phenomenal in its own way.

9.6.3 USING OTHER TESTING LIBRARIES

Testing has many facets, and in this book we've chosen to focus on unit and integration testing with Camel; these disciplines have the strongest coupling to Camel. Other disciplines, such as load and performance testing, are more generally applicable. You can find other resources that cover those concepts well. We do have a list of noteworthy testing libraries we want to bring to your attention.

AWAITILITY

Testing asynchronous systems is hard. Not only does it require handling threads, time-outs, and concurrency issues, but the intent of the test code can be obscured by all these details. Awaitility (<https://github.com/awaitility/awaitility>) is a DSL that allows you to express expectations of an asynchronous system in a concise and easy-to-read manner.

BYTEMAN

Byteman (<http://byteman.jboss.org>) is a bytecode manipulation tool that can inject custom code into specific points. For example, you can use Byteman to inject code that triggers errors to happen at certain places.

CITRUS INTEGRATION TESTING

Citrus (www.citrusframework.org) is a test framework that's intended to help you become successful with integration testing. With Citrus, you can build automated integration testing and have support for testing with Apache Camel. You can find a Camel and Citrus example with the source code in the chapter9/citrus directory.

JAVA MISSION CONTROL

Java Mission Control (JMC) is a part of Java that you can run from a shell. JMC is able to collect low-level details about running Java applications with little overhead (1% or less). With this tool, you can visually analyze the collected data to help you understand how your application is behaving on the JVM.

JMH

JMH (<http://openjdk.java.net/projects/code-tools/jmh>) is a Java harness for building, running, and analyzing micro benchmarks written in Java. With JMH, you set up a new Java project that can be created using a Maven archetype and you embed your application by adding the needed JARs/dependencies to the project. You then write benchmarks by writing small tests annotated with `@Benchmark`. You can easily specify to run tests in intervals of tens or thousands and also allow the JVM to warm up first.

GATLING

Gatling (<http://gatling.io>) is a load- and performance-testing

library that provides a DSL enabling you to write load testing at a high abstraction level. Gatling provides beautiful and informative reports as output of running the tests.

YOURKIT

YourKit (<https://yourkit.com>) is a Java CPU and memory profiler. With YourKit, you can sample running JVMs and gather statistics and reports. YourKit comes with a graphical application for visualizing hot spots in your application, such as the most-used methods, or methods that use the most CPU time or memory. The Camel team has used this tool with success to find performance bottlenecks in the Camel routing engine.

WIRESHARK

Wireshark (www.wireshark.org) is a network packet analyzer. It's used for network troubleshooting and inspection of network protocols. Wireshark comes with a graphical interface that lets you see what happens at the lower networking level.

That's all, folks. We've reached the end of this topic.

9.7 Summary and best practices

Testing is a challenge for every project. It's generally considered bad practice to perform testing only at the end of a project, so testing often begins when development starts and continues through the remainder of the project lifecycle.

Testing isn't something you do only once, either. It's involved in most phases in a project. You should do unit testing while you develop the application. And you should implement integration testing to ensure that the various components and systems work together. You also have the challenge of ensuring that you have the right environments for testing.

Camel can't eliminate these challenges, but it does provide a great Camel Test Kit that makes writing tests with Camel

applications easier and less time-consuming. You can run Camel in any kind of Java environment you like. We covered how to write Camel unit tests for the most popular ways of using Camel, whether it's running plain Camel standalone, with Spring in any form or shape, with CDI, OSGi, or on popular Java EE servers such as WildFly.

We also reviewed how to simulate real components using mocks in the earlier phases of a project, allowing you to test against systems you may not currently have access to. In chapter 11, you'll learn all about error handling; in this chapter you saw how to use the Camel Test Kit to test error handling by simulating errors.

We reviewed techniques for integration testing that don't involve using mocks. Doing integration testing, using live components and systems, is often harder than unit testing, in which mocks are a real advantage. In integration testing, mocks aren't available to use, so you have to use other techniques such as setting up a notification scheme that can notify you when certain messages have been processed. This allows you to inspect the various systems to see whether the messages were processed as expected (such as by peeking into a JMS queue or looking at a database table).

Toward the end of this chapter, we covered testing Camel with numerous integration testing libraries that make it possible to write tests that deploy and run in the application server of choice. You saw how Arquillian makes it easier to run tests inside a Java EE server such as WildFly. For OSGi users, we covered how to use Pax Exam for running tests inside Apache Karaf containers.

Here are a few best practices to take away from the chapter:

- *Use unit tests*—Use the Camel Test Kit from the beginning and write unit tests in your projects.
- *Use the mock component*—The mock component is a powerful component for unit testing. Use it rigorously in your unit tests.

- *Amend routes before testing*—Learn how to use the advice feature that allows you to manipulate your Camel routes before running your unit tests.
- *Test error handling*—Integration is difficult, and unexpected errors can occur. Use the techniques you've learned in this chapter to simulate errors and test that your application is capable of dealing with these failures.
- *Use integration tests*—Build and use integration tests to test that your application works when integrated with real and live systems.
- *Run tests on an application server*—If you run your Camel applications on an application server, we recommend using Arquillian or Pax Exam to run tests that deploy and run on the application server.
- *Test REST services*—The third-party testing library Rest Assured is a great tiny library for writing human-understandable unit tests that call REST services. Like Camel, it provides a DSL that almost speaks in human terms about what's happening. Make sure to pick up this library if you jump on the hype of REST and JSON.
- *Automate tests*—Strive for as much test automation as possible. Running your unit tests as well as integration tests should be as easy and repeatable as possible. Set up a continuous integration platform that performs testing on a regular basis and provides an instant feedback loop to your development team.

Speaking of REST, what a great segue to the next chapter, which is also a long chapter covering modern RESTful web services.

10

RESTful web services

This chapter covers

- Understanding RESTful web services fundamentals
- Building RESTful web services with JAX-RS and Apache CXF
- Building RESTful web services with camel-restlet and camel-cxf components
- Using Camel's Rest DSL
- Using Rest DSL with various Camel components
- Binding between POJOs and XML/JSON
- Documenting your RESTful services by using Swagger
- Browsing the Swagger API via the Swagger UI

For over a decade, web services have played an important role in integrating loose and disparate systems together. As a natural evolution of humankind and the IT industry, we learn and get smarter. Recently, RESTful web services (a.k.a. REST services) have taken over as the first choice for integrating services. This chapter covers a lot of ground about developing and using RESTful web services with Camel.

Section 10.1 covers the fundamentals and principles of RESTful services, so you're ready to tackle the content of this

chapter. We provide a lot of examples with source code that focus on a simple use-case that could happen in the real world, if Rider Auto Parts were an actual company. The section continues by explaining how to implement REST services using various REST frameworks and Camel components.

Section 10.2 introduces you to the Rest DSL, which allows you to define REST services *the Camel way*. You can use the natural verbs from REST together with the existing Camel routing DSL, combining both worlds.

Section 10.3 ends our REST trilogy by covering how to document your REST services by using Swagger APIs. This allows consumers to easily obtain contracts of your services.

Using SOAP web services with Camel

If you're looking for information about using SOAP web services with Apache Camel, this was covered in the first edition of this book in chapter 7 (section 7.4). You can find up-to-date source code examples of using SOAP web services in the `chapter10/code-first` and `chapter10/contract-first` directories that comes with this second edition of the book.

Okay, are you ready? Let's start with the new and existing stuff around RESTful services.

10.1 RESTful services

RESTful services, also known as *REST services*, has become a popular architectural style used in modern enterprise projects. REST was defined by Roy Fielding in 2000 when he published his paper, and today REST is a foundation that drives the modern APIs on the web. You can also think of it as a modern web service, in which the APIs are RESTful and HTTP based so

they're easily consumable on the web. This section uses an example from Rider Auto Parts to explain the principles of a RESTful API and then moves on to cover implementing this service by using the following REST frameworks and Camel components:

- *Apache CXF*—Using only CXF to build REST services
- *Apache CXF with Camel*—Same with Camel included
- *Camel Restlet component*—Using one of the simplest Camel REST components
- *Camel CXF REST component*—Using Camel with the CXF REST component

Before we start the show, let's cover the basic principles of a RESTful service.

10.1.1 UNDERSTANDING THE PRINCIPLES OF A RESTFUL API

At Rider Auto Parts, you've recently developed a web service that allows customers to retrieve information about their orders. The web service was previously implemented using old-school SOAP web services. Because you're a devoted developer, and because the company offered limited resources, you had to take matters into your own hands, and over a weekend you sketched the RESTful API design detailed in table [10.1](#).

Table 10.1 The RESTful API for the Rider Auto Parts order web service

Verb	http://rider.com/orders	http://rider.com/orders/{id}
GET	Retrieves a list of all the orders	Retrieves the order with the given ID
PUT	N/A	Updates the order with the given ID
POST	Creates a new order	N/A
DELETE	Cancels all the orders	Cancels the order with the given ID

The Rider Auto Parts order web service offers an API over HTTP

at the base path `http://rider.com/orders`. REST services reuse the same HTTP verbs that are already well understood and established from HTTP. The `GET` verb is like a SQL query function to access and return data. The `PUT` and `POST` verbs are similar to SQL `UPDATE` or `INSERT` functions that create or update data. And finally, the `DELETE` verb is for deleting data.

To allow legacy systems to access the service, you decide to support both XML and JSON as the data representation of the service.

We begin our REST journey with JAX-RS, which is a standard Java API for developing REST services.

10.1.2 USING JAX-RS WITH REST SERVICES

Java API for RESTful Web Services (JAX-RS) is a Java standard API that provides support for creating web services. Numerous REST frameworks implement the JAX-RS specification, such as Apache CXF, JBoss RESTEasy, Restlet, and Jersey. JAX-RS is primarily developed with the help of a set of Java annotations that you use in Java classes representing the web services.

Before we dive into JAX-RS, let's simplify the use case from Rider Auto Parts by removing two of the services from table [10.1](#) that aren't necessary. This leaves the four services listed in table [10.2](#).

Table 10.2 The first set of the RESTful API for the Rider Auto Parts order web service

Verb	<code>http://rider.com/orders</code>	<code>http://rider.com/orders/{id}</code>
GET	N/A	Retrieves the order with the given ID
PUT	N/A	Updates the order with the given ID
POST	Creates a new order	N/A
DELETE	N/A	Cancels the order with the given ID

You can then map the services from table [10.2](#) to a Java

interface:

```
public interface OrderService {  
    Order getOrder(int orderId);  
    void updateOrder(Order order);  
    String createOrder(Order order);  
    void cancelOrder(int orderId);  
}
```

And you create a Java model class to represent the order details:

```
@XmlRootElement(name = "order")  
@XmlAccessorType(XmlAccessType.FIELD)  
public class Order {  
    @XmlElement  
    private int id;  
    @XmlElement  
    private String partName;  
    @XmlElement  
    private int amount;  
    @XmlElement  
    private String customerName;  
  
    // getter/setter omitted  
}
```

The model class is annotated with JAXB annotations, making message translation between Java and XML/JSON much easier, because frameworks such as JAXB and Jackson know how to map payloads between XML/JSON and the Java model, and vice versa.

The actual REST service is defined in a JAX-RS resource class implementation, as shown in the following listing.

Listing 10.1 JAX-RS implementation of the Rider Auto Parts REST service

```
import javax.ws.rs.DELETE; ①
```

1

Java imports the JAX-RS annotations

```
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;
```

```
@Path("/orders/") 2
```

2

Defines the entry path of the RESTful web service

```
@Produces("application/xml")
public class RestOrderService {

    private OrderService orderService;

    public void setOrderService(OrderService orderService)
{ 3 }
```

3

To inject the OrderService implementation

```
        this.orderService = orderService;
    }

    @GET
    @Path("/{id}")
    public Response getOrder(@PathParam("id") int orderId)
{ 4 }
```

4

GET order by ID

```
        Order order = orderService.getOrder(orderId);
        if (order != null) {
            return Response.ok(order).build(); 5
```

5

The Response builder to return the order model in the response

```
    } else {
        return
Response.status(Response.Status.NOT_FOUND).build();
    }
}

@PUT
public Response updateOrder(Order order) { 6
```

6

PUT to update an existing order

```
orderService.updateOrder(order);
return Response.ok().build();
}

@POST
public Response createOrder(Order order) { 7
```

7

POST to create a new order

```
String id = orderService.createOrder(order);
return Response.ok(id).build();
}

@DELETE
@Path("/{id}")
public Response cancelOrder(@PathParam("id") int
orderId) { 8
```

8

DELETE to cancel an existing order

```
orderService.cancelOrder(orderId);
return Response.ok().build();
}
```

At first, the class imports JAX-RS annotations ❶. A class is defined as a JAX-RS resource class by annotating the class with `@Path` ❷. The `@Path` annotation on the class level defines the entry of the context-path the service is using. In this example, all the REST services in the class must have their context-path begin with `/orders/`. The class is also annotated with `@Produces("application/xml")` ❸, which declares that the service outputs the response as XML. You'll later see how to use JSON or support both.

Because you want to separate the JAX-RS implementation from your business logic, you use dependency injection to inject the `orderService` instance ❹ that holds the business logic.

The service that gets orders by ID is annotated with `@Path("/{id}")` ❺, as well as in the method signature using `@PathParam("id")`. The parameter is mapped to the context-path with the given key. If a client calls the REST service with `/orders/123`, for example, then `123` is resolved as the ID path parameter, which will be mapped to the corresponding method parameter.

If the service returns an order, notice that the response builder includes the order instance when the `ok` response is being created ❻. And, likewise, if no order could be found, the response builder is used to create an HTTP 404 (Resource Not Found) error response.

The two services to update ❼ and create ⬽ an order are just two lines of code. But notice that these two methods use the `Order` type as a parameter. And because you've annotated the `Order` type with JAXB annotations, Apache CXF will automatically map the incoming XML payload to the `Order` POJO class.

The last method to cancel an order by ID ⬾ uses similar path parameter mapping as the get orders by ID service.

The book's source code contains this example in the `chapter10/cxf-rest` directory, and you can try the example by

using the following Maven goal:

```
mvn compile exec:java
```

The example includes two dummy orders, which makes it easier to try the example. For example, to get the first order, you can open the following URL from a web browser:

```
http://localhost:9000/orders/1
```

The second order, not surprisingly, has the order ID of 2, so you can get it using the following:

```
http://localhost:9000/orders/2
```

Testing REST services from a web browser

Testing REST services from a web browser is easy only when testing GET services, as those can be accessed by typing the URL in the address bar. But to try POST, PUT, DELETE, and other REST verbs is much harder. Therefore, you can install third-party plugins that provide a REST client. Google Chrome, for example, has the popular Postman plugin.

USING JSON INSTEAD OF XML

Apache CXF allows you to plug in various binding providers, so you can plug in Jackson for JSON support. To do this, you need to add the following two Maven dependencies to the pom.xml file:

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxfrs-rt-rs-extension-providers</artifactId>
    <version>3.2.1 </version>
</dependency>
<dependency>
```

```
<groupId>com.fasterxml.jackson.jaxrs</groupId>
<artifactId>jackson-jaxrs-json-provider</artifactId>
<version>2.8.10</version>
</dependency>
```

Sadly, CXF doesn't support autodiscovering the providers from the classpath, so you need to enlist the Jackson provider on the CXF RS server:

```
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
sf.setProvider(JacksonJsonProvider.class);
```

The chapter10/cxf-rest-json directory in the book's source code contains this example, which you can try using the following Maven goal:

```
mvn compile exec:java
```

Apache CXF vs. Jersey

Both Apache CXF and Jersey are web frameworks that support RESTful web services. Apache Camel had first-class integration with CXF for a long time, provided by the camel-cxf component. Numerous people working on Apache Camel are also CXF committers. At the time of this writing, Camel has no Jersey component, and there are no such plans. Apache CXF is also a sister Apache project and therefore recommended to use over Jersey.

In this section, we've covered using Apache CXF without Camel. Because this is a book about Apache Camel, it's time to get back on the saddle and ride. Let's see how to add Camel to the Rider Auto Parts order service application in the sections to follow. But first, we'll summarize the pros and cons of using a pure Apache CXF model for REST services in table [10.3](#).

Table 10.3 Pros and cons of using a pure CXF-RS approach for REST services

Pros	Cons
------	------

Uses the JAX-RS standard.	Not possible to define REST services without having to write Java code.
Apache CXF-RS is a well-established REST library.	Camel isn't integrated first class.
The REST service allows developers full control of what happens as they can change the source code.	Configuring CXF-RS can be nontrivial. There's no <i>CXF in Action</i> book.

Okay, let's try to add Camel to the example.

10.1.3 USING CAMEL IN AN EXISTING JAX-RS APPLICATION

You managed to develop the Rider Auto Parts order application by using JAX-RS with CXF. But you know that down the road, changes are in the pipeline that require integration with other systems. For that, you want Camel to play its part. How can you develop REST services by using the JAX-RS standard and still use Camel? One way is to let CXF integrate with Camel.

Using Camel from any existing Java application can be done in many ways. Possibly one of the easiest is to use Camel's `ProducerTemplate` from the Java application to send messages to Camel. The following listing shows the modified JAX-RS service implementation that uses Camel.

[Listing 10.2](#) Modified JAX-RS REST service that uses Camel's `ProducerTemplate`

```
@Path("/orders/")
@Produces("application/json")
public class RestOrderService {

    private ProducerTemplate producer;

    public void setProducerTemplate(ProducerTemplate
producerTemplate) { ①
```

1

To inject Camel's ProducerTemplate

```
        this.producer = producerTemplate;
    }

    @GET
    @Path("/{id}")
    public Response getOrder(@PathParam("id") int orderId)
{
    Order order =
producer.requestBody("direct:getOrder",
                     orderId,
Order.class);    2
```

2

Calls Camel route to get the order by ID

```
    if (order != null) {
        return Response.ok(order).build();
    } else {
        return
Response.status(Response.Status.NOT_FOUND).build();
    }
}

@PUT
public Response updateOrder(Order order) {
    producer.sendBody("direct:updateOrder",
order);    3
```

3

Calls Camel route to update order

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

@POST
public Response createOrder(Order order) {
    String id =
producer.requestBody("direct:createOrder",
                     order,
String.class);    4
```

4

Calls Camel route to create order

```
        return Response.ok(id).build();  
    }  
  
    @DELETE  
    @Path("/{id}")  
    public Response cancelOrder(@PathParam("id") int  
orderID) {  
        producer.sendBody("direct:cancelOrder",  
orderID); 5  
    }
```

5

Calls Camel route to cancel order

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

This code looks similar to the code in [listing 10.1](#), but differs in that Camel's ProducerTemplate **1** will be used to route messages from the REST web service to a Camel route (**2** **3** **4** **5**). The Camel route is responsible for routing the message to the Rider Auto Parts back-end system, giving access to the order details.

Because you have both CXF and Camel independently operating in the same application, you need to set them individually and configure them in the right order. To keep things simple, you decide to run the application as a standalone Java application with a single main class to bootstrap the application, as shown in the following listing.

[Listing 10.3](#) Running CXF REST web service with Camel integrated from Main class

```
public class RestOrderServer {  
  
    public static void main(String[] args) throws Exception  
{  
    OrderRoute route = new OrderRoute(); 1  
}
```

1

Camel route that accesses the back-end order system

```
route.setOrderService(new BackendOrderService());  
  
CamelContext camel = new  
DefaultCamelContext(); 2
```

2

Creates CamelContext and adds the route

```
camel.addRoutes(route);  
  
ProducerTemplate producer =  
camel.createProducerTemplate(); 3
```

3

Creates ProducerTemplate

```
RestOrderService rest = new RestOrderService();  
rest.setProducerTemplate(producer); 4
```

4

Injects ProducerTemplate to REST web service

```
JAXRSServerFactoryBean sf = new  
JAXRSServerFactoryBean(); 5
```

5

Sets up CXF REST web service

```
sf.setResourceClasses(RestOrderService.class); 5  
  
sf.setResourceProvider(RestOrderService.class,  
                      new  
SingletonResourceProvider(rest)); 5  
sf.setProvider(JacksonJsonProvider.class); 6
```

6

Uses Jackson for JSON support

```
sf.setAddress("http://localhost:9000/"); 7
```

7

The URL for published REST web service

```
Server server = sf.create();  
camel.start(); 8
```

8

Starts Camel and CXF

```
server.start();  
// code that keeps the JVM running omitted 9
```

9

Keeps the JVM running until it's stopped

```
camel.stop(); 1
```

10

Stops Camel and CXF

```
    server.stop();  
}  
}
```

The first part of the application is to set up Camel by creating the route ① used to call the back-end system. Then Camel itself is created with CamelContext ②, where the route is added. To integrate Camel with CXF, you create ProducerTemplate ③ that gets injected into the REST resource class ④. After setting up

Camel, it's CXF's turn. To use a JAX-RS server in CXF, you create and configure it by using `JAXRSServerFactoryBean` ⑤. Support for JSON is provided by adding the Jackson provider ⑥. Then the URL for the entry point for clients to call the service is configured ⑦ as the last part of the configuration. The application is started by starting Camel and CXF in that order ⑧. To keep the application going, some code keeps the JVM running until it's signaled to stop ⑨. To let the application stop gracefully, CXF and Camel are stopped ⑩ before the JVM terminates.

TIP You can also set up CXF REST web services by using an XML configuration, which we cover later in this chapter.

This example is shipped as part of the book's source code in the `chapter10/cxf-rest-camel` directory; you can run the example by using the following Maven goal:

```
mvn compile exec:java
```

Combining CXF with Camel gives you the best of two worlds. You can define and code the JAX-RS REST services by using the JAX-RS standard in Java code, and still integrate Camel by using a simple dependency-injection technique to inject `ProducerTemplate` or a Camel proxy. But it comes with the cost of having to write this in Java code, and manually having to control the lifecycle of both CXF and Camel. These pros and cons are summarized in table 10.4.

Table 10.4 Pros and cons of using a CXF-RS with Camel approach

Pros	Cons
Uses the JAX-RS standard.	Not possible to define REST services without having to write Java code.
Apache CXF-RS is a well-established REST library.	Configuring CXF-RS can be nontrivial.

The REST service allows developers full control of what happens as they can change the source code.	Need to set up both CXF and Camel lifecycle separately.
Camel is integrated using dependency injection into the JAX-RS resource class.	There's no <i>CXF in Action</i> book.

Later you'll see how to use CXF REST services from a pure Camel approach, in which Camel handles the lifecycle of CXF by using the camel-cxf component. We'll leave CXF for a moment and look at alternative REST libraries you can use. For example, you'll learn how the service can be implemented without JAX-RS, and using only Camel with the camel-restlet component.

10.1.4 USING CAMEL-RESTLET WITH REST SERVICES

The camel-restlet component is one of the easiest components to use for hosting REST services as Camel routes. The component has also been part of Apache Camel for a long time (it was included at the end of 2008). Let's see it in action.

To implement the services from table 10.2 as REST services with camel-restlet, all you need to do is define each service as a Camel route and call the order service bean that performs the action, as shown in the following listing.

Listing 10.4 Camel route using camel-restlet to define four REST services

```
public class OrderRoute extends RouteBuilder {
    public void configure() throws Exception {
        from("restlet:http://0.0.0.0:8080/orders?
restletMethod=POST") ①
```

①

REST service to create order

```
.bean("orderService", "createOrder");
```

```
from("restlet:http://0.0.0.0:8080/orders/{id}?
restletMethod=GET") ②
```

②

REST service to get order by ID

```
.bean("orderService", "getOrder(${header.id})");

from("restlet:http://0.0.0.0:8080/orders?
restletMethod=PUT") ③
```

③

REST service to update order

```
.bean("orderService", "updateOrder");

from("restlet:http://0.0.0.0:8080/orders/{id}?
restletMethod=DELETE") ④
```

④

REST service to cancel order by ID

```
.bean("orderService", "cancelOrder(${header.id})");
}
```

These REST web services are defined with a route per REST operation, so there are four routes in total. Notice that the restlet component uses the `restletMethod` option to specify the HTTP verb in use, such as `POST` ①, `GET` ②, `PUT` ③, and `DELETE` ④. The restlet component allows you to map dynamic values from the context-path by using the `{ }` style, which you use to map the ID from context-path to a header on the Camel message, as illustrated in [figure 10.1](#).

```
from("restlet:http://0.0.0.0:8080/orders/{id}?restletMethod=GET")
```

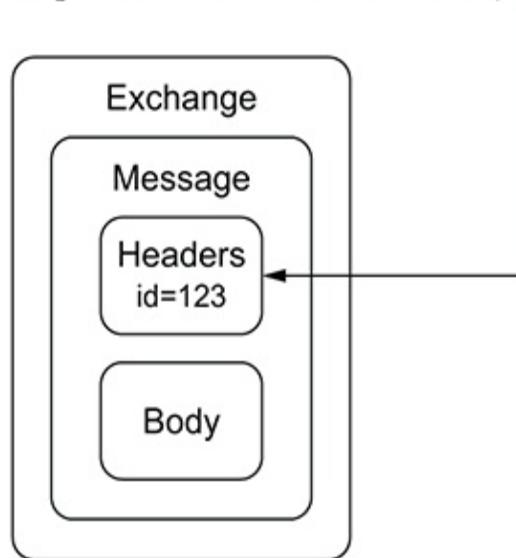


Figure 10.1 Mapping of dynamic values from context-path to Camel message headers with the Camel Restlet component

In [figure 10.1](#), the REST web service to get the order by ID from [listing 10.4](#) is shown at the top. Each dynamic value in the context-path declared by `{key}` is mapped to a corresponding Camel message header when Camel routes the incoming REST call.

It's important to not mix this syntax with Camel's property placeholder. The mapping syntax uses a single `{ }` pair, whereas Camel's property placeholder uses double `{}{ }` pairs.

Rest DSL vs. EIP DSL

The routes in [listing 10.4](#) use the normal Java DSL syntax: routes begin with `from` followed by `bean` or `to` with the EIP verbs. Later in this chapter, you'll learn about the Rest DSL that allows you to define REST web services by using REST verbs (GET, POST, PUT, and so on) instead of the EIP (`from`, `to`, `bean`, and so on) verbs.

You can try this example, which is available in the book's source

code in the chapter10/restlet directory, by using the following Maven goals:

```
mvn test -Dtest=OrderServiceTest  
mvn test -Dtest=SpringOrderServiceTest
```

The restlet component performs no automatic binding between XML or JSON to Java objects, so you'd need to add camel-jaxb or camel-jackson to the classpath to provide this.

BINDING XML TO/FROM POJOs USING CAMEL-JAXB

The example in chapter10/restlet supports XML binding by including camel-jaxb as a dependency. By adding camel-jaxb to the classpath, a *fallback type converter* is added to Camel's type-converter registry. A fallback type converter is capable of converting between multiple types decided at runtime, whereas regular type converters can convert from only one type to another. By using a fallback type converter, Camel is able to determine at runtime whether converting to/from a POJO is possible if the POJO has been annotated with JAXB annotations.

When you want to use JSON, you need to use camel-jackson.

BINDING JSON TO/FROM POJOs USING CAMEL-JACKSON

What camel-jaxb does for converting between XML and POJOs is similar to what camel-jackson does. But a few extra steps are required when using camel-jackson:

- Enable camel-jackson as a fallback type converter
- Optionally annotate the POJOs with JAXB annotations
- Optionally annotate the POJOs with Jackson annotations

Both camel-jaxb and camel-jackson can use the same set of JAXB annotations, which allow you to customize the way the mapping to your POJOs should happen. But Jackson can perform mapping to any POJOs if the POJO has plain getter/setters, and the JSON payload matches the names of

those getter/setter methods. You can use JAXB to further configure the mapping, and Jackson provides annotations on top of that. You'll find more details on the Jackson website: <https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations>.

To enable camel-jackson as a fallback type converter, you need to configure properties on `CamelContext`. From Java code, you do as follows:

```
context.getProperties().put("CamelJacksonEnableTypeConverter", "true");
context.getProperties().put("CamelJacksonTypeConverterToPojo", "true");
```

You need to configure this before you start `CamelContext`—for example, from the `RouteBuilder` where you set up your Camel routes. When using XML, you need to configure this inside

```
<camelContext>
```

```
<camelContext
  xmlns="http://camel.apache.org/schema/spring">
  <properties>
    <property key="CamelJacksonEnableTypeConverter"
      value="true"/>
    <property key="CamelJacksonTypeConverterToPojo"
      value="true"/>
  </properties>
  ...
</camelContext>
```

The accompanying source code includes an example of this in the `chapter10/restlet-json` directory; you can try the example by using the following Maven goals:

```
mvn test -Dtest=OrderServiceTest
mvn test -Dtest=SpringOrderServiceTest
```

TIP camel-restlet can also be used to call a REST service from a Camel route (for example, as a producer). As an example, see the source code of the `chapter10/restlet-json` example, where the unit test uses `ProducerTemplate` with a camel-restlet URI to call

the REST web service.

The camel-restlet component was one of the first Camel components to integrate with REST. The component is easy to use and requires little configuration to get going. The pros/cons of using camel-restlet have been summarized in table [10.5](#).

Table 10.5 Pros and cons of using camel-restlet

Pros	Cons
Easy to get started.	Doesn't use the JAX-RS standard.
Restlet is an established REST library.	camel-restlet hasn't been integrated with Google Gson or Jackson to make using JSON easy.

Let's look at one more REST library before ending this section. We'll circle back to Apache CXF and take a look at the Camel CXF-RS component.

10.1.5 USING CAMEL-CXF WITH REST SERVICES

The camel-cxf component includes support for both REST and SOAP web services. As we noted previously, the difference between using plain Apache CXF and the camel-cxf component is that the latter is Camel *first*, whereas CXF is CXF *first*. By *first*, we mean that either CXF or Camel is the primary driver behind how the wheel is spinning.

When using camel-cxf, you need to configure the CXF REST server, which can be done in two ways:

- Camel-configured endpoint
- CXF-configured bean

CAMEL-CONFIGURED ENDPOINT

Using a Camel-configured endpoint means that you configure

the CXF REST server as a regular Camel endpoint by using URI notation, as shown in the following listing.

Listing 10.5 Camel route using camel-cxf to define four REST services

```
public class OrderRoute extends RouteBuilder {  
  
    public void configure() throws Exception {  
        from("cxfrs:http://localhost:8080  
            ?resourceClasses=camelinaction.RestOrderService  
            &bindingStyle=SimpleConsumer")  
            1  
    }  
}
```

1

Camel-configured endpoint of REST service using CXF-RS

```
.toD("direct:${header.operationName}");  
    2
```

2

Calls the route that should perform the selected REST operation

```
from("direct:createOrder")  
    3
```

3

REST service to create order

```
.bean("orderService", "createOrder");
```

```
from("direct:getOrder")  
    4
```

4

REST service to get order by ID

```
.bean("orderService", "getOrder(${header.id}))";
```

```
from("direct:updateOrder")  
    5
```

5

REST service to update order

```
.bean("orderService", "updateOrder");  
from("direct:cancelOrder") 6
```

6

REST service to cancel order by ID

```
.bean("orderService", "cancelOrder(${header.id})");  
}  
}
```

When using camel-cxf, the REST web service is defined in a single route **①** where you configure the hostname and port the REST server uses. When using camel-cxf, it's recommended to use the `SimpleConsumer` binding style, which binds to the natural JAX-RS types and Camel headers. The default binding style uses the `org.apache.cxf.message.MessageContentsList` type from CXF, which is a lower-level type like Camel's Exchange. The last configured option is the `resourceClasses` option **②**, which refers to a class or interface that declares the REST web service by using the JAX-RS standard.

Using a class or interface for the resource class

Because CXF uses JAX-RS, the REST web service is defined as a JAX-RS resource class that lists all the REST operations. The resource class code is in [listing 10.6](#). The resource class is, in fact, not a class but an interface; the reason for this is that camel-cxf will use JAX-RS only as a contract to set up REST web services. At runtime, the incoming request is routed in the Camel routes instead of invoking the Java code in the resource class. And because of that, it's more appropriate to use an interface instead of a class with empty methods. Another reason to use an

interface over a class is that the class would have all empty methods and not be in use. This may lead developers to become confused about why this class is empty and to think that it could be a mistake or a bug.

The REST web service has four services (as shown in [listing 10.6](#)), and the header indicates which operation was called by using the name `operationName`. You use this to route the message via the Dynamic To EIP pattern ②, whereby you let each operation be a separate route (③ ④ ⑤ ⑥), linked together using the direct component.

The actual REST web service is specified as a JAX-RS resource class, shown in the following listing.

Listing 10.6 JAX-RS interface of the Rider Auto Parts REST service

```
@Path("/orders")
@Consumes(value = "application/xml,application/json")
@Produces(value = "application/xml,application/json")
public interface RestOrderService { ①
```

①

Using an interface instead of a class

```
@GET
@Path("/{id}")
Order getOrder(@PathParam("id") int orderId); ②
```

②

REST operations

```
@PUT
void updateOrder(Order order); ②
```

```
@POST
String createOrder(Order order); ②
```

```
@DELETE  
@Path("/{id}")  
void cancelOrder(@PathParam("id") int orderId); ②  
}
```

In this example, you use an interface ①, for the reason explained in the preceding sidebar. Each REST operation is declared using standard JAX-RS annotations ② that allow you to define the REST web services in a nice, natural way. Notice that the return types in the operations are the model types (such as `order`, which returns the order details). And the `createOrder` operation returns a `String` with the ID of the created order.

Now compare [listing 10.6](#) with [listing 10.1](#). In [listing 10.6](#), you can see from the interface what the REST service operations return, such as an `Order` type in the `getOrder` operation, and a `String` type in the `createOrder` operation. In [listing 10.1](#), all the REST service operations return the same JAX-RS Response type. When doing so, you lose the ability to know exactly what the operation returns.

This problem can be remedied using a third-party API documentation framework such as Swagger. Swagger provides a set of annotations you can use in the JAX-RS resource classes to specify all sorts of details, such as what the REST operations take as input parameters, and what they return. In section 10.3, you'll learn how to use Swagger with Camel.

You can try this example, available in the `chapter10/camel-cxf-rest` directory of this book's source code, using the following Maven goals:

```
mvn test -Dtest=RestOrderServiceTest  
mvn test -Dtest=SpringRestOrderServiceTest
```

You can also configure the CXF REST server by using a bean style, as explained next.

CXF-CONFIGURED BEAN

Apache CXF was created many years ago, when the Spring

Framework was popular and using Spring was dominated by XML configuration. This affected CXF, as the natural way of configuring CXF is the Spring XML style. The *CXF-configured bean* is such a style.

To use camel-cxf with Spring XML (or Blueprint XML), you need to add the camel-cxf namespace to the XML stanza (highlighted in bold). This allows you to set up the CXF REST server by using the `rsServer` element, as shown in the following listing.

Listing 10.7 Configuring CXF REST server using XML style

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

<bean id="orderService"
      class="camelinaction.DummyOrderService"/>

<cxf:rsServer id="restOrderServer"
      address="http://localhost:8080" 1>
```

1

Sets up CXF REST server using `rsServer` element

```
serviceClass="camelinaction.RestOrderService">
</cxf:rsServer>

<camelContext id="camel"
```

```
xmlns="http://camel.apache.org/schema/spring">
<route>
    <from uri="cxfrs:bean:restOrderServer?
bindingStyle=SimpleConsumer"/> ②
```

Camel route with the CXF-configured bean

```
<toD uri="direct:${header.operationName}"/> ③
```

Calling the route that should perform the selected REST operation

```
</route>
<route>
    <from uri="direct:createOrder"/>
    <bean ref="orderService" method="createOrder"/>
</route>
<route>
    <from uri="direct:getOrder"/>
    <bean ref="orderService"
method="getOrder(${header.id})"/>
</route>
<route>
    <from uri="direct:updateOrder"/>
    <bean ref="orderService" method="updateOrder"/>
</route>
<route>
    <from uri="direct:cancelOrder"/>
    <bean ref="orderService"
method="cancelOrder(${header.id})"/>
</route>
</camelContext>
</beans>
```

The CXF REST server is configured in the `rsserver` element **①** to set up the base URL of the REST service and to refer to the resource class. The `rsserver` allows additional configuration such as logging, security, payload providers, and much more. All this configuration is done within the `rsserver` element and uses standard CXF naming. You can find many examples and details on the Apache CXF website.

The routes within `<camelContext>` are the XML equivalent of the Java-based example in [listing 10.5](#).

This example is provided as source code in the chapter10/camel-cxf-rest directory; you can try the example using the following Maven goal:

```
mvn test -Dtest=SpringRestBeanOrderServiceTest
```

The example uses XML as a payload during testing, but you can also support JSON.

ADDING JSON SUPPORT TO CAMEL-CXF

JSON is supported in CXF by different providers. The default JSON provider is Jettison, but today the most popular JSON library is Jackson. To use Jackson, all you have to do is add the following Maven dependencies to your pom.xml file:

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-rs-extension-providers</artifactId>
    <version>3.2.1</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.jaxrs</groupId>
    <artifactId>jackson-jaxrs-json-provider</artifactId>
    <version>2.8.10</version>
</dependency>
```

In the Spring or Blueprint XML file, add the Jackson provider as a bean:

```
<bean id="jsonProvider"
    class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider"
/>
```

And finally, add the provider to the CXF REST server configuration:

```
<cxfrsServer id="restOrderServer"
    address="http://localhost:8080"
    serviceClass="camelaction.RestOrderService">
    <cxfrs:providers>
        <ref bean="jsonProvider"/>
    </cxfrs:providers>
</cxfrsServer>
```

```
</cxf:providers>  
</cxf:rsServer>
```

We provide an example with the book's source code in the chapter10/camel-cxf-rest directory. You can try the example using the following Maven goal:

```
mvn test -Dtest=SpringRestBeanOrderServiceJSonTest
```

Using CDI, Apache Karaf, or Spring Boot

The example is also available for running on CDI, Karaf, or Spring Boot, which you can find in the chapter10/camel-cxf-rest-cdi, chapter10/camel-cxf-rest-karaf, and chapter10/camel-cxf-rest-spring-boot directories, respectively. Each example has instructions for running it in the accompanying readme file.

Table 10.6 details the pros and cons of using camel-cxf.

Table 10.6 Pros and cons of using camel-cxf for REST services

Pros	Cons
Uses the JAX-RS standard.	Not possible to define REST services without having to write Java code.
Apache CXF-RS is a well-established REST library. CXF-RS is integrated with Camel as first class.	The code in the REST service isn't executed, and developers are allowed full control only of what happens in Camel routes.
	There's no <i>CXF in Action</i> book.

That was a lot of coverage on using various REST frameworks with Camel. You saw how Camel bridges to REST services by

using Camel components and Camel routes. It's as if the REST and EIP worlds are separated. What if you could take the REST verbs into the EIP world so you could define REST services by using HTTP verbs in the same style as your EIPs in Camel routes? This is the topic of the next section.

10.2 The Camel Rest DSL

Several REST frameworks have emerged and gained huge popularity recently—for example, the Ruby-based web framework called Sinatra, the Node-based Express framework, and from Java the small REST and web framework called Spark Java. What they all share is putting a Rest DSL in front of the end user, making it easy to do REST development.

The story of the Rest DSL all began with conversations between James Strachan and Claus Ibsen, discussing how making REST services with Apache Camel wasn't as straightforward as they'd like it to be. For example, the semantics of the REST and HTTP verbs became *less visible* in the Camel endpoints. What if we could define REST services using a *REST-like* DSL in Camel? And so it began. Claus started hacking on the Rest DSL, and now two years later, he writes about it in this book.

Rest DSL design

The goal of Rest DSL is to make defining REST services at a high abstraction level easy and quick, and to do it *the Camel way*. The Rest DSL is intended to support 95% of use cases. For some advanced use cases, you may have to opt out of the Rest DSL and use an alternative solution, such as using JAX-RS directly with Apache CXF or Jersey. Initially, the Rest DSL supported only exposing RESTful services from Camel (for example, as a consumer) and there was no support for calling existing external RESTful

services. But from Camel 2.19 onward, the Rest DSL has preliminary support for calling RESTful services as a producer. This chapter focuses on the consumer side and has only limited coverage of the producer side.

The Rest DSL debuted at end of 2014 as part of Apache Camel 2.14. But it has taken a couple of releases to further harden and improve the Rest DSL, making it a great feature in the Camel toolbox.

Before covering all the ins and outs of the Rest DSL, let's see it in action.

10.2.1 EXPLORING A QUICK EXAMPLE OF THE REST DSL

The Camel Rest DSL was inspired by the Java Spark REST library, and so camel-spark-rest was the first component that's fully Rest DSL integrated. Before the Rest DSL, camel-restlet probably came closest to using the REST and HTTP verbs in the endpoint URIs. Take a moment to look at the camel-restlet example in [listing 10.4](#) and then compare it to the implementation of a similar solution with the Rest DSL in the following listing. You should notice that the latter *says more* with REST and HTTP terms.

Listing 10.8 Camel route using Rest DSL to define four REST services

```
public class OrderRoute extends RouteBuilder {  
    public void configure() throws Exception {  
        restConfiguration()  
            .component("spark-rest").port(8080); 1  
    }  
}
```

1

Configures Rest DSL to use camel-spark-rest component

```
rest("/orders") ②
```

②

Uses REST verbs to define the four REST services with GET, POST, PUT and DELETE

```
.get("{id}")
    .to("bean:orderService?
method=getOrder(${header.id})")
    .post()
        .to("bean:orderService?method=createOrder")
    .put()
        .to("bean:orderService?method=updateOrder")
    .delete("{id}")
        .to("bean:orderService?
method=cancelOrder(${header.id})");
}
}
```

When using the Rest DSL, it must be configured first; you need to indicate which REST component to use ① and additional configurations such as the context-path and port number. In this example, you haven't configured any context-path, so it uses / by default. Then you define a set of REST services by using `rest("/orders")` ② followed by each service defined using REST verbs such as get, post, put, and delete. The four REST services will at runtime be mapped to the following URLs:

- GET /orders/{id}
- POST /orders
- PUT /orders
- DELETE /orders/{id}

Dynamic values in context-path can be mapped by using the { } syntax, which is done in the case of get and delete.

You can also use the Rest DSL in the XML DSL, as shown in the following listing.

Listing 10.9 Camel route using Rest DSL to define four REST services using XML DSL

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
    <restConfiguration component="spark-rest"
port="8080"/> 1
```

1

Configures Rest DSL to use camel-spark-rest component

```
<rest path="/orders"> 2
```

2

Uses REST verbs to define the four REST services with GET, POST, PUT and DELETE

```
<get uri="{id}">
    <to uri="bean:orderService?
method=getOrder(${header.id})"/>
</get>
<post>
    <to uri="bean:orderService?method=createOrder"/>
</post>
<put>
    <to uri="bean:orderService?method=updateOrder"/>
</put>
<delete uri="{id}">
    <to uri="bean:orderService?
method=cancelOrder(${header.id})"/>
</delete>
</rest>
</camelContext>
```

As you can see, the Rest DSL in Java or XML DSL is structured in the same way. At first you need to configure the Rest DSL **1**, followed by defining the REST services by using the REST verbs **2**.

You can try this example, provided as part of the source code for the book, by running the following Maven goals in the

chapter10/spark-rest directory:

```
mvn test -Dtest=OrderServiceTest  
mvn test -Dtest=SpringOrderServiceTest
```

That's it for a quick example of the Rest DSL in action. Let's return to Camel in action and talk about how the Rest DSL works under the covers and which Camel components it supports.

10.2.2 UNDERSTANDING HOW THE REST DSL WORKS

In a nutshell, the Rest DSL works as a DSL extension (to the existing Camel routing DSL), using REST and HTTP verbs, that's then mapped to the existing Camel DSL. By mapping to the Camel DSL, it's all just routes from the Camel point of view.

This is better explained with an example, so let's take the `GET` service from listing 10.8

```
<rest path="/orders">  
  <get uri="{id}">  
    <to uri="bean:orderService?  
method=getOrder(${header.id})"/>  
  </get>  
  ...  
</rest>
```

and see how it's mapped to the Camel routing DSL:

```
<route>  
  <from uri="rest:get:/orders/{id}?componentName=spark-  
rest"/>  
  <to uri="bean:orderService?  
method=getOrder(${header.id})"/>  
  ...  
</route>
```

As you can see, this route is just a regular Camel route with a `<from>` and a `<to>`. The meat of the mapping happens between `<get>` and `<from>`:

```
<rest path="/orders">  
"rest:get:/orders/{id}?"  
  <get uri="{id}"> →
```

```
componentName=spark-rest"
```

You can break down the mapping as illustrated in figure 10.2.

```
rest:get:/orders/{id}?componentName=spark-rest
```

① ② ③ ④ ⑤

Figure 10.2 Key points of mapping the Rest DSL to five individual parts in the Camel rest endpoint

The key points are as follows:

1. rest—Every REST service is mapped to use the generic Camel rest component, which is provided out of the box in camel-core.
2. get—The HTTP verb, such as GET, POST, or PUT.

3 /orders—The base URL of the REST services, configured in `<rest path="/orders">`.

4 /{id}—Additional URI of the REST service, configured in `<get uri="{id}">`.

5. componentName=spark-rest—The Camel component to use for hosting the REST services. Additional parameters can be passed on from the REST configuration.

Notice that points 3 and 4 combine the base URL with the URI of the REST service.

The mapping for the four REST services is as follows:

```
rest:get:/orders/{id}?componentName=spark-rest
rest:post:/orders?componentName=spark-rest
rest:put:/orders?componentName=spark-rest
rest:delete:/orders/{id}?componentName=spark-rest
```

You've now established that the Rest DSL is glorified syntax sugar on top of the existing DSL that rewrites the REST services as small Camel routes of `from → to`. REST services are just Camel routes, which means all the existing capabilities in Camel are being harnessed. This also means that managing these REST

services becomes as easy as managing your existing Camel routes—yes, you can start and stop those REST routes, and so on.

Rest DSL = Camel routes

To emphasize one more time: the Rest DSL builds regular Camel routes, which are run at runtime. So you have nothing to be afraid of. All you've learned about Camel routes is usable with the Rest DSL.

If the Rest DSL is just Camel routes, what's being used to set up an HTTP server and servicing the requests from clients calling the REST services?

10.2.3 USING SUPPORTED COMPONENTS FOR THE REST DSL

When using Rest DSL, you need to pick one of the Rest DSL–capable components that handle the task of hosting the REST services. At the time of this writing, the following components support Rest DSL:

- *camel-jetty*—Using the Jetty HTTP server
- *camel-netty4-http*—Using the Netty library
- *camel-restlet*—Using the Restlet library
- *camel-servlet*—Using Java Servlet for servlets or Java EE servers
- *camel-spark-rest*—Using the Java Spark REST library
- *camel-undertow*—Using the JBoss Undertow HTTP server

You may be surprised that the list contains components that aren't primarily known as REST components. Only *camel-restlet* and *camel-spark-rest* are REST libraries; the remainder are typical HTTP components. This is on purpose, as the Rest DSL was designed to be able to sit on top of any HTTP component

and just work (famous last words). A prominent component absent from the list is Apache CXF. At the time of this writing, CXF is too tightly coupled to the programming model of JAX-RS, requiring the REST services to be implemented as JAX-RS service classes. This might change in the future, as some Apache Camel and CXF committers want to work on this so CXF can support Camel's Rest DSL.

With plenty of choices in the list of components, you can pick and choose what best suits your use case. For example, you can use camel-servlet if you run your application in a servlet container such as Apache Tomcat or WildFly. You can also “go serverless” and run your Camel REST applications standalone with Jetty, Restlet, Spark Rest (uses Jetty), Undertow, or the Netty library.

Why do you need to use one of these components with the Rest DSL? Because you don't want to reinvent the wheel. To host REST services, you need an HTTP server framework, and it's much better to use an existing one than to build your own inside Camel. The Rest DSL has very little code to deal with HTTP and REST, but leaves that entirely up to the chosen component.

The Rest DSL makes it easy to switch between these components. All you have to do is change the component name in the REST configuration. But that's not entirely true, as all these components have their own configuration you can use. For example, you'd need to do this to set up security and other advanced options.

10.2.4 CONFIGURING REST DSL

Configuring Rest DSL can be grouped into the following six categories:

- *Common*—Common options
- *Component*—To configure options on the chosen Camel component to be used
- *Endpoint*—To configure endpoint-specific options from the

chosen component on the endpoint that are used to create the consumer in the Camel route hosting the REST service

- *Consumer*—To configure consumer-specific options in the Camel route that hosts the REST service
- *Data format*—To configure data-format-specific options that are used for XML and JSON binding
- *CORS headers*—To configure CORS headers to include in the HTTP responses

We cover the first four categories next. The data format category is explained in section 10.2.5, and the CORS headers category is covered in section 10.3 as part of API documentation of your REST services using Swagger.

CONFIGURING COMMON OPTIONS

The common options are listed in table 10.7.

Table 10.7 Common options for the Rest DSL

Option	Default	Description
component		<i>Mandatory.</i> The Camel component to use as the HTTP server. The chosen name should be one of the supported components listed in section 10.2.3.
scheme	http	Whether to use HTTP or HTTPS.
hostname		The hostname for the HTTP server. If not specified, the hostname is resolved according to the <code>restHostNameResolver</code> .
port		The port number to use for the HTTP server. If you use the servlet component, the port number isn't in use, as the servlet container controls the port number. For example, in Apache Tomcat the default port is 8080, and in Apache Karaf / ServiceMix, it's 8181.

contextPath	Configures a base context-path that the HTTP server will use.
restHostNameResolver	Resolves the hostname to use, if no hostname has been explicitly configured. You can specify localHostName OR localIp to use either the hostname or IP address.

All the REST configuration uses restConfiguration or <restConfiguration>, as shown here in Java:

```
restConfiguration()
    .component("spark-
rest").contextPath("/myservices").port(8080);
```

And in XML:

```
<restConfiguration component="spark-rest"
    contextPath="/myservices" port="8080"/>
```

You can try this yourself. For example, let's change the example in chapter10/spark-rest to use another component such as camel-undertow. All you have to do is change the dependency in pom.xml from camel-spark-rest to camel-undertow and change the component name to undertow as shown here:

Here's the changed pom.xml file:

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

And here's the change in the OrderRoute Java source code:

```
restConfiguration()
    .component("undertow").port(8080);
```

Also remember to change the component in the SpringOrderServiceTest.xml file:

```
<restConfiguration component="undertow" port="8080"/>
```

Then run the example by using the following Maven goal in the chapter10/spark-rest directory:

```
mvn clean test
```

When using Rest DSL, it uses the default settings of the chosen Camel component. You can configure this on three levels: the component, the endpoint, and the consumer. We'll now take a look at how to do that.

CONFIGURING COMPONENT, ENDPOINT, AND CONSUMER OPTIONS IN REST DSL

Camel components often offer many options you can tweak to your needs. We covered using components in chapter 6, which explains how configuration works. The Rest DSL allows you to configure the component directly in the REST configuration section on three levels:

- Component level
- Endpoint level
- Consumer level

The component level isn't used as often as the endpoint level. As you should know this far into the book, Camel relies heavily on endpoints that you configure as URI parameters. Therefore, you'll find most of the options at this level. The consumer level is seldom in use, as consumers are configured from endpoint options. [Table 10.8](#) shows the option name to use in Rest DSL to refer to these levels.

Table 10.8 Component-related configuration options for the Rest DSL

Option	Description
componentProperty	To configure component-specific options. You can use multiple options to specify more than one option.
endpointProperty	To configure endpoint-specific options. You can use multiple options to specify more than one option.

consumerProperty

To configure consumer-specific options. You can use multiple options to specify more than one option.

A common use-case for the need to configure Rest DSL on components is to set up security or configure thread pool settings. At Rider Auto Parts, the RESTful order application you've been working on has a new requirement for clients to authenticate before they can access the services. Because Jetty supports HTTP basic authentication, you choose to give it a go with Jetty. In addition, the load on the application is expected to be minimal, so you want to reduce the number of threads used by Jetty. All this is configured on the Jetty component level, which is easily done with the Rest DSL, as shown here:

```
restConfiguration()
    .component("jetty").port(8080)
        .componentProperty("minThreads", "1")
        .componentProperty("maxThreads", "8");
```

And in XML DSL:

```
<restConfiguration component="jetty" port="8080">
    <componentProperty key="minThread" value="1"/>
    <componentProperty key="maxThread" value="8"/>
</restConfiguration>
```

Notice how easy that is, by using `componentProperty` to specify the option by key and value. In the preceding example, you lower the Jetty thread pool to use only one to eight worker threads to service incoming calls from clients.

Setting up security with Jetty requires much more work, depending on the nature of the security. Security in Jetty is implemented using special handlers, which in our case is `ConstraintSecurityHandler`. You create and configure this handler in the `JettySecurity` class, as shown in the following listing.

[Listing 10.10](#) Set up Jetty authentication using basic auth

```
public class JettySecurity {
```

```
public static ConstraintSecurityHandler  
createSecurityHandler() {  
    Constraint constraint = new Constraint("BASIC",  
"customer"); 1
```

1

Uses HTTP Basic Auth

```
constraint.setAuthenticate(true);  
  
ConstraintMapping mapping = new ConstraintMapping();  
mapping.setConstraint(constraint);  
mapping.setPathSpec("/*"); 2
```

2

Applies to all incoming requests matching /*

```
ConstraintSecurityHandler handler = new  
ConstraintSecurityHandler();  
handler.addConstraintMapping(mapping);  
handler.setAuthenticator(new BasicAuthenticator());  
handler.setLoginService(new  
HashLoginService("RiderAutoParts",  
"src/main/resources/users.properties")); 3
```

3

Loads user names from external file

```
    return handler;  
}
```

To use HTTP basic authentication, you specify BASIC as a parameter to the created Constraint object **1**. Then you tell Jetty to apply the security settings to the incoming requests that match the pattern **2**, which means all of them. The last part is to put all this together in a handler object that's returned. The known username and passwords are validated using

`LoginService`. Rider Auto Parts keeps it simple by using a plain-text file to store the username and passwords ③—well, at least for this prototype. Later, when you go into production, you'll have to switch to an LDAP-based `LoginModule` instead.

The last piece of this puzzle is to configure Rest DSL to use this security handler, which is done on the endpoint level instead of the component level. The REST configuration is updated as shown here:

```
restConfiguration()
    .component("jetty").port(8080)
        .componentProperty("minThreads", "1")
        .componentProperty("maxThreads", "8")
    .endpointProperty("handlers", "#securityHandler");
```

And in XML DSL:

```
<restConfiguration component="jetty" port="8080">
    <componentProperty key="minThread" value="1"/>
    <componentProperty key="maxThread" value="8"/>
    <endpointProperty key="handlers"
value="#securityHandler"/>
</restConfiguration>
```

As you can see, Jetty uses the `handlers` option on the endpoint level to refer to one or more Jetty security handlers. Notice that you use the `#` prefix in the value to tell Camel to look up the security handler from the Camel registry.

TIP Chapter 4 covers the Camel registries.

All that's left to do is create an instance of the handler by calling the static method `createSecurityHandler` on the `JettySecurity` class. For example, in Spring XML, you can do this:

```
<bean id="securityHandler"
class="camelinaction.JettySecurity"
factory-method="createSecurityHandler"/>
```

The book's source code contains this example in `chapter10/jetty-`

rest-security, and you can try the example using the following Maven goals:

```
mvn compile exec:java
```

From a web browser, you can access <http://localhost:8080/orders/1>, which should present a login box. You can pass in `jack` as the username and `123` as the password.

The example also provides unit tests, which you can run with the following Maven goals:

```
mvn test -Dtest=OrderServiceTest  
mvn test -Dtest=SpringOrderServiceTest
```

Using CDI or Apache Karaf

The example is also available for running on CDI or Karaf, which you can find in the `chapter10/jetty-rest-security-cdi` and `chapter10/jetty-rest-security-karaf` directories. Each example has instructions in the accompanying `readme` file.

When working with REST services, it's common to use data formats in XML or JSON. The following section covers how to use XML and JSON with the Rest DSL.

10.2.5 USING XML AND JSON DATA FORMATS WITH REST DSL

The Rest DSL supports automatic binding of XML/JSON content to/from POJOs using Camel data formats. You may want to use binding if you develop POJOs that map to your REST services request and response types. This allows you, as a developer, to work with the POJOs in Java code.

Rest DSL supports the binding modes listed in table [10.9](#).

Table 10.9 Binding modes supported by Rest DSL

Mode	Description
off	Binding is turned off. This is the default mode.
auto	Binding is automatically enabled if the necessary data format is available on the classpath. For example, providing camel-jaxb on the classpath enables support for XML binding. Having camel-jackson on the classpath enables support for JSON binding.
json	Binding to/from JSON is enabled and requires a JSON-capable data format on the classpath, such as camel-jackson.
xml	Binding to/from XML is enabled and requires camel-jaxb on the classpath.
json-xml	Binding to/from JSON and XML is enabled and requires both data formats to be on the classpath.

By default, the binding mode is off. No automatic binding occurs for incoming or outgoing messages. Section 10.2.2 covered how the Rest DSL works by mapping the Rest DSL into Camel routes. When binding is enabled, a `RestBindingProcessor` is injected into each Camel route as the first processor in the routing tree. This ensures that any incoming messages can be automatically bound from XML/JSON to POJO. Likewise, when the Camel route is completed, the binding is able to convert the message body back from POJO to XML/JSON.

Figure 10.3 illustrates this principle.

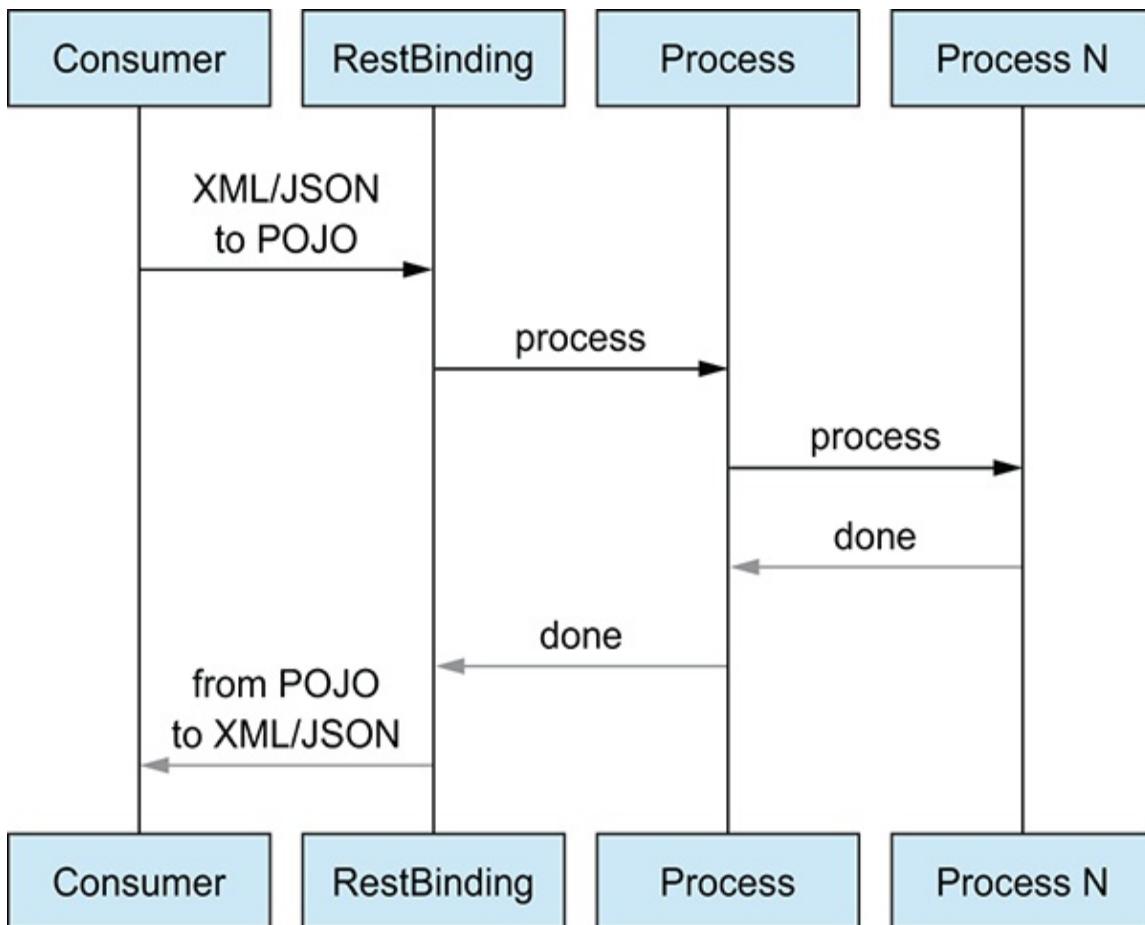


Figure 10.3 RestBinding sits right after the consumer to ensure that it's able to bind the incoming messages firsthand from XML/JSON to POJO classes, before the message is routed in the Camel route tree by the processors. When the routing is done, the outgoing message is reversed, from POJO to XML/JSON format, right before the consumer takes over and sends the outgoing message to the client that called the REST service.

The Rest DSL binding is designed to make it easy to support the two most common data formats with REST services, XML and JSON. As a rule of thumb, you configure the binding mode to be XML, JSON, or both, depending on your needs. Then you add the necessary dependencies to the classpath, and you're finished.

How this works, as illustrated in [figure 10.3](#), is achieved from the RestBinding-Processor woven into each of the Camel routes that processes the REST services. This happens for any supported REST component (listed in section 10.2.3), and therefore you have binding out of the box.

Avoid double binding

Some REST libraries, such as Apache CXF and Restlet, provide their own binding support as well. Section 10.1.2 covered using Apache CXF with JAX-RS using JSON. In those use-cases, it's not necessary to enable their binding support as well, because it's provided out of the box with Camel's Rest DSL.

`RestBindingProcessor` is responsible for handling content negotiation of the incoming and outgoing messages, which depends on various factors.

BINDING NEGOTIATION FOR INCOMING AND OUTGOING MESSAGES

`RestBindingProcessor` adheres to the following rules (in order of importance) about whether XML/JSON binding to POJO is applied for the *incoming* messages:

1. If binding mode is off, no binding takes place.
2. If the incoming message carries a `Content-Type` header, the value of the header is used to determine whether the incoming message is in XML or JSON format.
3. If the Rest DSL has been configured with a `consumes` option, that information is used to detect whether binding from XML or JSON format is allowed.
4. If you haven't yet determined whether the incoming message is in XML or JSON format, and the binding mode allows both XML and JSON, then the processor will check whether the message payload contains XML content. If the content isn't XML, it's assumed to be JSON. (At the time of writing, the Rest DSL binding supports only XML or JSON.)
5. If the binding mode and the incoming message are incompatible, an exception is thrown.

For *outgoing* messages, the rule set is almost identical:

1. If the binding mode is off, no binding takes place.
2. If the outgoing message carries an `Accept` header, the value of the header is used to determine whether the client accepts the message in either XML or JSON format.
3. If the outgoing message doesn't carry an `Accept` header, the `Content-Type` header is checked to see whether the message body is either in XML or JSON format.
4. If the Rest DSL has been configured with a `produces` option, that information is used to detect whether binding to XML or JSON format is expected.
5. If the binding mode and the outgoing message are incompatible, an exception is thrown.

The rules can also be explained as a way of ensuring that the configured binding mode and the actual incoming or outgoing message *align* so the binding can be carried on.

It's almost time to see this in action, but first you need to know how to configure the binding.

CONFIGURING REST DSL BINDING

Table 10.10 lists the binding options.

Table 10.10 Binding options for Rest DSL

Option	Default	Description
<code>bindingMode</code>	<code>off</code>	To enable binding where the Rest DSL can automatically bind the incoming and outgoing payload to XML or JSON format. The possible choices are <code>off</code> , <code>auto</code> , <code>json</code> , <code>xml</code> , and <code>json_xml</code> . The modes are further detailed in <u>table 10.9</u> .
<code>skipBindingOnErrorCode</code>	<code>true</code>	Whether to use binding when returning an error as the response. This allows you to build custom error messages that don't bind to JSON/XML, as success messages otherwise will do.

jsonDataFormat	json	Name of specific data format to use for JSON binding.
xmlDataFormat	jaxb	Name of specific XML data format to use for XML binding.
dataFormatProperties		Allows you to configure the XML and JSON data format with data-format-specific options. For example, when you need to enable JSON pretty-print mode.

The binding configuration is done using `restConfiguration` in Java:

```
restConfiguration()
    .component("spark-rest").port(8080)
    .bindingMode(RestBindingMode.json)
    .dataFormatProperty("prettyPrint", "true");
```

And in XML:

```
<restConfiguration component="spark-rest" port="8080"
bindingMode="json">
    <dataFormatProperty key="prettyPrint" value="true"/>
</restConfiguration>
```

Here you've configured the binding mode to be JSON only. And to specify that the JSON output should be in pretty-print mode, you configure this by using the `dataFormatProperty` option.

Configuring data format options

The REST configuration uses the option

`dataFormatProperty` to configure the XML and JSON data formats. The default XML and JSON data formats are `camel-jaxb` and `camel-jackson`. Both happen to provide an option named `prettyPrint` to turn on outputting XML/JSON in pretty-print mode.

When we started covering the Rest DSL in section 10.2.1, the first example used the `camel-spark-rest` component, and XML was the supported content-type of the REST services. Let's see what it takes to improve this example by using JSON instead, and then thereafter supporting both XML and JSON.

USING JSON BINDING WITH REST DSL

First you need to add the JSON data format by adding the following dependency to the Maven pom.xml file:

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

Then you need to enable JSON binding in the REST configuration, and you also turn on pretty-print mode so the JSON output from Camel is structured in a nice, human-readable style. The following listing shows the code in Java.

[Listing 10.11](#) Using Rest DSL with JSON binding with Java DSL

```
restConfiguration()
  .component("spark-rest").port(8080)
  .bindingMode(RestBindingMode.json) ①
```

①

Using JSON binding mode

```
  .dataFormatProperty("prettyPrint", "true"); ②
```

2

Turns on pretty printing for JSON output

```
rest("/orders")
    .get("/{id}").outType(Order.class)      3
```

3

The output type of this REST service is Order class.

```
    .to("bean:orderService?method=getOrder(${header.id})")
    .post().type(Order.class)      4
```

4

The input type of this REST service is Order class.

```
    .to("bean:orderService?method=createOrder")
    .put().type(Order.class)      5
```

5

The input type of this REST service is Order class.

```
    .to("bean:orderService?method=updateOrder")
    .delete("/{id}")
    .to("bean:orderService?
method=cancelOrder(${header.id})");
```

In the REST configuration, you set up to use the camel-spark-rest component and host the REST services on port 8080. The binding mode is set to JSON **1**, and you want the JSON output to be pretty printed **2** (using indents and new lines) instead of outputting all the JSON data on one line only.

When you use JSON binding mode, the incoming and outgoing messages are in JSON format. For example, the REST service to create an order **4** could receive the following JSON payload in an HTTP POST call:

```
{  
    "partName": "motor",
```

```
    "amount": 1,  
    "customerName": "honda"  
}
```

Because you've enabled JSON binding mode, the `RestBindingProcessor` needs to transform the incoming JSON payload to a POJO class. The JSON payload doesn't carry any kind of identifier that can be used to know which POJO class to use. Therefore, the Rest DSL needs to be marked with the class name of the POJO to use. This is done by using `type(class)`④ ⑤. The REST service to return an order can likewise be marked with the outgoing type by using `outType(class)` ③.

In this example, the input and output types are a single entity. The Rest DSL also supports using array/list types, which you specify as follows in Java DSL:

```
put().type(Order[].class)  
get("{id}").outType(Order[].class)
```

In XML DSL, you must prepend the name with `[]`, as highlighted here:

```
<put type="camelaction.Order[]>  
<get uri="{id}" outType="camelaction.Order[]>
```

TIP It's a good practice to mark the incoming and outgoing types in the Rest DSL, which is part of documenting your APIs. You'll learn much more about this in section 10.3, when Swagger enters the stage.

Readers who cheer for XML shouldn't be left out, so we prepared the following listing with the XML version.

[Listing 10.12](#) Using Rest DSL with JSON binding with XML DSL

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

```
<restConfiguration component="spark-rest" port="8080"  
bindingMode="json"> 1
```

1

Using JSON binding mode

```
    <dataFormatProperty key="prettyPrint"  
value="true"/> 2
```

2

Turns on pretty printing for JSON output

```
</restConfiguration>
```

```
<rest path="/orders">  
    <get uri="{id}" outType="camelaction.Order"> 3
```

3

The output type of this REST service is Order class.

```
        <to uri="bean:orderService?  
method=getOrder(${header.id})"/>  
        </get>  
        <post type="camelaction.Order"> 4
```

4

The input type of this REST service is Order class.

```
        <to uri="bean:orderService?method=createOrder"/>  
        </post>  
        <put type="camelaction.Order"> 5
```

5

The input type of this REST service is Order class.

```
        <to uri="bean:orderService?method=updateOrder"/>  
        </put>  
        <delete uri="{id}">  
            <to uri="bean:orderService?  
method=cancelOrder(${header.id})"/>
```

```
        </delete>
    </rest>

</camelContext>
```

That would be the changes needed for using JSON binding. This example is provided with the source code in the chapter10/spark-rest-json directory. You can try this example using the following Maven goals:

```
mvn test -Dtest=OrderServiceTest
mvn test -Dtest=SpringOrderServiceTest
```

Now let's make the example support both XML and JSON.

USING BOTH XML AND JSON BINDING WITH REST DSL

We continue the example and turn on both JSON and XML binding, which is done by having camel-jackson and camel-jaxb as dependencies in the Maven pom.xml file. To demonstrate how easy it is to switch the REST component, you change camel-spark-rest to camel-undertow instead. Therefore, the updated pom.xml file should include the following dependencies:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-undertow</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jackson</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jaxb</artifactId>
</dependency>
```

The only changes you then have to make are to reconfigure the REST configuration to use undertow and both XML and JSON binding, as shown here:

```
restConfiguration()
    .component("undertow").port(8080)
    .bindingMode(RestBindingMode.json_xml)
```

```
.dataFormatProperty("prettyPrint", "true");
```

The book's source code contains this example in the chapter10/undertow-rest-xml-json directory, and you can try the example by running the following Maven goal:

```
mvn compile exec:java
```

Then from the command line, you can use curl (or wget) to call the REST service, such as getting the order number 1:

```
curl http://localhost:8080/orders/1
```

This returns the response in JSON format:

```
$ curl http://localhost:8080/orders/1
{
  "id" : 1,
  "partName" : "motor",
  "amount" : 1,
  "customerName" : "honda"
}
```

Now to get the response in XML instead, you need to provide the HTTP Accept header and specify to accept XML content:

```
$ curl --header "Accept: application/xml"
http://localhost:8080/orders/1
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<order>
  <id>1</id>
  <partName>motor</partName>
  <amount>1</amount>
  <customerName>honda</customerName>
</order>
```

That seems easy enough. The client can now specify the acceptable output format, and the Camel Rest DSL binding makes all that happen with little for the developer to do. Well, hold your horses—er, camels. When doing data transformation between XML and POJO with JAXB, you need to do one of the following:

- Annotate the POJO classes with @XmlElement

- Provide an `ObjectFactory` class

The former is by far the easiest and most-used approach but requires you to add annotations to all your POJO classes. The latter lets you create an `ObjectFactory` class in the same package where the POJO class resides. Then the `ObjectFactory` class has methods to create the POJO classes and be able to control how to map to the fields. We, the authors of this book, favor using annotations.

TIP You can use JAXB annotations on your POJO classes, which both JAXB and Jackson can use. But Jackson has its own set of annotations you can use to control the mapping between JSON and POJO.

USING SPRING BOOT CONFIGURATION WITH REST DSL

Spring Boot users can also configure rest configuration in the `application.properties` or `application.yaml` file. The previous example can be configured as follows:

```
camel.rest.component=undertow
camel.rest.port=8080
camel.rest.binding-mode=json
camel.rest.data-format-property.prettyPrint=true
```

We have provided an example in the `chapter10/springboot-json` directory which you can try by running the following Maven goal:

```
mvn spring-boot:run
```

From a web browser you can open
`http://localhost:8080/api/orders/1`.

Because the example is running in Spring Boot, it's recommended to use the servlet engine from Spring Boot instead of undertow, which is done by specifying `servlet` as the component name:

```
camel.rest.component=servlet
```

But you can often omit configuring this with Spring Boot because Camel is able to automatically detect that camel-servlet is on the classpath and use it. We recommend you take a look at this example for more details.

We'll now leave the REST binding and talk about one last item when using Rest DSL: how do you deal with exceptions?

HANDLING EXCEPTIONS AND RETURNING CUSTOM HTTP STATUS CODES WITH REST DSL

Integration is hard, especially the unhappy path where things fail and go wrong. This book devotes all of chapter 11 to this subject. In this section, you'll learn how to use the existing error-handling capabilities with Camel and your Rest DSL services.

At Rider Auto Parts, the order service should deal with failures as well, and you've amended the service to throw exceptions in case of failures:

```
public interface OrderService {  
    Order getOrder(int orderId) throws  
OrderNotFoundException;  
    void updateOrder(Order order) throws  
OrderInvalidException;  
    String createOrder(Order order) throws  
OrderInvalidException;  
    void cancelOrder(int orderId);  
}
```

We've introduced two new exceptions. `OrderNotFoundException` is thrown if a client tries to get an order that doesn't exist. In that situation, you want to return an HTTP status code of 404, which means Resource Not Found. `OrderInvalidException` is thrown when a client tries to either create or update an order and the input data is invalid. In that situation, you want to return a status code 400 to the client. Finally, any other kind of exception is regarded as an internal server error, and an HTTP status code 500 should be returned.

How can you implement such a solution? Luckily, as you'll see in chapter 11 on error-handling, Camel allows you to handle exceptions by using `onException` in the DSL. You spend less than 10 minutes adding the error handling to the `OrderRoute` class that contains the Rest DSL. The following listing shows the REST configuration, error handling, and REST services all together.

Listing 10.13 Using `onException` to handle exceptions and return HTTP status codes

```
restConfiguration()
    .component("undertow").port(8080)
    .bindingMode(RestBindingMode.json_xml)
    .dataFormatProperty("prettyPrint", "true");

onException(OrderNotFoundException.class) ①
```

1

Handles `OrderNotFoundException` and returns HTTP status 404 with empty body

```
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(404))
    .setBody(constant(""));
```

```
onException(OrderInvalidException.class) ②
```

2

Handles `OrderInvalidException` and returns HTTP status 400 with empty body

```
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(400))
    .setBody(constant(""));
```

```
onException(Exception.class) ③
```

3

Handles all other exceptions and returns HTTP status 500 with exception message in the HTTP body

```

    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(500))
    .setBody(simple("${exception.message}\n"));

rest("/orders")
    .get("/{id}").outType(Order.class)
        .to("bean:orderService?method=getOrder(${header.id})")
    .post().type(Order.class)
        .to("bean:orderService?method=createOrder")
    .put().type(Order.class)
        .to("bean:orderService?method=updateOrder")
    .delete("/{id}")
        .to("bean:orderService?
method=cancelOrder(${header.id})");

```

As you can see, all you did was add three `onException` blocks ❶ ❷ ❸ to catch the various exceptions and then set the appropriate HTTP status code and HTTP body.

The book's source code contains this example in the `chapter10/undertow-rest-xml-json` directory. We prepared scenarios to demonstrate the three exception handlers used. At first, start the example using the following Maven command:

```
mvn compile exec:java
```

For example, to try to call the GET service to get an order that doesn't exist, you can run the following command, which returns HTTP status code 404:

```
$ curl -i --header "Accept: application/json"
http://localhost:8080/orders/99
HTTP/1.1 404 Not Found
```

Another example is to update an order using invalid input data:

```
$ curl -i -X PUT -d @invalid-update.json
http://localhost:8080/orders --header "Content-Type:
application/json"
HTTP/1.1 400 Bad Request
```

And you've also prepared a special request to trigger a server-side error:

```
$ curl -i -X POST -d @kaboom-create.json
```

```
http://localhost:8080/orders --header "Content-Type:  
application/json"  
HTTP/1.1 500 Internal Server Error  
...  
Forced error due to kaboom
```

One last feature we want to quickly cover is the latest addition to the Rest DSL: being able to call RESTful services.

10.2.6 CALLING RESTFUL SERVICES USING REST DSL

From Camel 2.19 onward, the Rest DSL introduced the first functionality to allow Camel to easily call any RESTful service by using the rest component. The rest component builds on the principle of the Rest DSL by using REST verbs and URI templating.

Suppose you want to call an existing REST service that provides a simple API to return geographical information about a given city or country. Such an external REST service can quickly be implemented using Spring Boot and Camel together:

```
@RestController  
public class GeoRestController {  
  
    @EndpointInject  
    private FluentProducerTemplate template;  
  
    @RequestMapping("/country/{city}") ①
```

①

Defines Spring REST service

```
        public Object address(@PathVariable(name = "city") String  
        city) {  
            return template.to("geocoder:address:" +  
        city).request(); ②
```

②

Calls Camels geocoder component

```
    }  
}
```

The REST service has one REST endpoint ❶ that maps to `/country/{city}`. You then use Camel's geocoder component to obtain geolocation information about the city ❷. This component will call an online service on the internet, which returns detailed information in JSON format.

Now you want to use Camel's rest component to call the REST service. To make this easy, you use a timer to trigger the Camel route periodically.

Listing 10.14 Using the rest component to call the REST service

```
restConfiguration().producerComponent("http4") ❶  
  
❶ Configures Rest DSL producer-side  
  
.host("localhost").port(8080); ❶  
  
from("timer:foo?period=5000")  
    .setHeader("city", RestProducerRoute::randomCity) ❷  
  
❷ Sets a random city  
  
.to("rest:get:country/{city}") ❸  
  
❸ Calls rest component  
  
.transform().jsonpath("$.results[0].formattedAddress") ❹  
  
❹
```

Extracts formatted response

```
.log("${body}");
```

When using Rest DSL to call a REST service, you also need to configure it. Listing 10.14 specifies that you want to use the http4 component as the HTTP client that will perform the REST calls ①. You also specify the hostname and port number of the REST service ①. If you omit this information, you'd need to provide it in the rest endpoint ③. But if you call the same host, it's recommended that you configure this in the rest configuration (as done in the listing). The example then generates a random city name ②, which will be used when calling the REST service. The REST service is then called using the URI

rest:get:country/{city}. The rest component supports URI templating, which means the URI at runtime will be resolved to `http://localhost:8080/country/Paris` if the random city is Paris, and an HTTP GET operation is executed. The information returned is detailed, and you want to log only a short, human-readable message, so you use jsonpath to extract a formatted address ④, which is then logged.

This example is provided with the source code in the chapter10/rest-producer directory, and you can try it by using this:

```
mvn spring-boot:run
```

Notice the console output, which should print information about Paris, London, and Copenhagen.

Sorry, but that's all we managed to cover about the new producer side of the Rest DSL. Apache Camel will improve the Rest DSL in its upcoming releases and make working with REST services even easier. A rest-swagger component enables you to call a REST service by referring to operation IDs, if the REST service has Swagger API documentation associated. We also managed to add a new Maven plugin (camel-restdsl-swagger) that allows you to generate Camel Rest DSL Java source code based on an existing Swagger API documentation. These new

additions are expected to be improved over the upcoming release, so we suggest you take a look at this when you get a chance.

Okay, you've now covered a lot about Rest DSL, but there's more to come. The next section covers APIs and how to document your REST services directly in the Rest DSL by using Swagger.

10.3 API documentation using Swagger

Any kind of integration between service providers and clients (or *consumers* and *producers* in Camel lingo) requires using a data format both parties can use. RESTful services using JSON, or to a lesser extent XML, are by far the popular choices. But this is only one part of the puzzle: the way the data is structured in the payload.

To complete the jigsaw puzzle, you need pieces that address technical areas, such as the following:

- What services does a service producer offer?
- Where are the endpoints to access these services?
- What data format do the services accept?
- What data format can the services respond with?
- What's the mandatory and optional information to provide when calling a service?
- And where should said information be provided—as HTTP headers, query parameters, in the payload?
- What does the service do?
- What error codes can the service return?
- Is the service secured?
- Is there a little example?

And besides the technical questions, as human beings we'd like to know the following:

- What does the service do?
- What do the parameters mean?
- What do incoming and outgoing payloads represent?

As you probably can tell, we could add many other questions. But it all leads to the same picture. How do we document these services, so both service producers and clients can interact successfully?

WSDL and WADL

SOAP-based web services have service contracts built in from the Web Services Description Language (WSDL) specification. This specification describes the functionality offered by the web services. For RESTful services, an attempt was made to ratify Web Application Description Language (WADL) as an official specification, but that didn't happen. In practice, writing WSDL or WADL schema files by hand is problematic for developers, even for developers trying to read and comprehend existing schema files. Therefore, most often, tools are used to machine-generate schema files based on parsing source code, known as the *source-first approach*. Both WSDL and WADL were designed by a committee and lacked what users wanted. The open source communities, aiming to build something better, brought other projects to life, and out of those, one stood on top: Swagger.

SWAGGER TO THE RESCUE

Swagger is both a specification and framework for describing, producing, consuming, and visualizing RESTful services.

Applications written with the Swagger framework contain full documentation of methods, parameters, and models directly in the source code. This ensures that the implementation and documentation of RESTful services are always in sync.

In 2016, the Swagger specification was renamed the *OpenAPI* specification and governed by its founding group under the Linux Foundation. In this chapter, we use the popular term *Swagger* instead of *OpenAPI*.

The Swagger specification is best explained by quoting the OpenAPI website:

The goal of the OAI specification is to define a standard, language-agnostic interface to REST APIs that allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interfaces have done for lower-level programming, Swagger removes the guesswork in calling the service.

[—https://openapis.org/specification](https://openapis.org/specification)

This section covers getting started with Swagger and documenting a JAX-RS REST service. You'll see how to provide Swagger documentation directly in the Rest DSL to easily document your REST services. We'll finish the Swagger coverage by showing you how to embed the popular Swagger UI into your Camel applications.

10.3.1 USING SWAGGER WITH JAX-RS REST SERVICES

Swagger provides a set of annotations you can use to document JAX-RS-based REST services and model classes. The annotations are fairly easy to use, but adding annotations to all your REST operations, parameters, and model classes is tedious.

To start using Swagger annotations, you add the following Maven dependency to your pom.xml file:

```
<dependency>
  <groupId>io.swagger</groupId>
  <artifactId>swagger-jaxrs</artifactId>
  <version>1.5.16</version>
</dependency>
```

The following listing shows the Order Service example from Rider Auto Parts, which has been documented using the Swagger annotations.

Listing 10.15 Using Swagger annotations to document JAX-RS REST service

```
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import io.swagger.annotations.ApiParam;
import io.swagger.annotations.ApiResponse;
import io.swagger.annotations.ApiResponses; 1
```

1

Imports Swagger annotations

```
@Path("/orders/")
@Consumes("application/json,application/xml")
@Produces("application/json,application/xml")
@Api(value = "/orders", description = "Rider Auto Parts
Order Service") 2
```

2

Documents the REST API

```
public class RestOrderService {

    @GET
    @Path("/{id}")
    @ApiOperation(value = "Get order", response =
Order.class) 3
```

3

Documents each REST operation

```
@ApiResponses({ 5
```

5

Documents response of REST operation

```
    @ApiResponse(code = 200, response = String.class, 5  
                  message = "The found order"), 5  
    @ApiResponse(code = 404, response = String.class, 5  
                  message = "Cannot find order with the  
id")) 5  
    public Response getOrder(@ApiParam(value = "The id of the  
order", 4
```

4

Documents each REST input parameter

```
                    required = true)  
@PathParam("id") int orderId) {  
    Order order = orderService.getOrder(orderId);  
    if (order != null) {  
        return Response.ok(order).build();  
    } else {  
        return  
Response.status(Response.Status.NOT_FOUND).build();  
    }  
}
```

```
@PUT
```

```
@ApiOperation(value = "Update existing order") 3
```

3

Documents each REST operation

```
    public Response updateOrder(@ApiParam(value = "The order  
to update", 4
```

4

Documents each REST input parameter

```
        required = true) Order order)
{
    orderService.updateOrder(order);
    return Response.ok().build();
}

@POST
@ApiOperation(value = "Create new order") 3
```

3

Documents each REST operation

```
@ApiResponses({@ApiResponse(code = 200, response =
String.class, 5
```

5

Documents response of REST operation

```
        message = "The id of the created order"))
public Response createOrder(@ApiParam(value = "The order
to create", 4
```

4

Documents each REST input parameter

```
        required = true) Order order)
{
    String id = orderService.createOrder(order);
    return Response.ok(id).build();
}

@DELETE
@Path("/{id}")
public Response cancelOrder(@ApiParam(value = "The order
id to cancel", 4
```

4

Documents each REST input parameter

```
        required = true) @PathParam("id")
```

```
    int orderId) {
        orderService.cancelOrder(orderId);
        return Response.ok().build();
    }
}
```

To document JAX-RS services using Swagger, you add the Swagger annotations to the JAX-RS resource class. At first, you import the needed Swagger annotations ❶. The `@Api` annotation ❷ is used as high-level documentation of the services in this class. In this example, they're the order services. Each REST operation is then documented using `@ApiOperation` ❸; you supply a description of what the operation does as well as what the operation returns. Each operation that takes input parameters is documented using `@ApiParam` ❹, which is colocated with the parameter. Swagger allows you to further specify response values using `@ApiResponse` ❺. In this example, you use this to specify that the `createOrder` operation returns the ID of the created order, in the success operation. Also notice that ❻ documents both the successful and error response from the `getOrder` operation. Each `@ApiResponse` must map to an HTTP response code: `200` means success, for example, and `404` means resource not found.

The model classes can be annotated with `@ApiModelProperty`, as shown in the following listing.

Listing 10.16 Document model classes using Swagger annotations

```
@XmlRootElement(name = "order")
@XmlAccessorType(XmlAccessType.FIELD)
@ApiModelProperty(value = "order", description = "Details of the
order") ❶
```

❶

Documents the model class

```
public class Order {
```

```
@XmlElement  
@ApiModelProperty(value = "The order id", required =  
true) ②
```

②

Documents each field in the model class

```
private int id;  
  
@XmlElement  
@ApiModelProperty(value = "The name of the part",  
required = true) ②  
private String partName;  
  
@XmlElement  
@ApiModelProperty(value = "Number of items ordered",  
required = true) ②  
private int amount;  
  
@XmlElement  
@ApiModelProperty(value = "Name of the customer",  
required = true) ②  
private String customerName;  
  
// getter/setters omitted  
}
```

Documenting model classes is easier than JAX-RS classes because you need only to document each field ② and the overall model ①.

After adding all the annotations to your JAX-RS and model classes, you need to integrate Swagger with CXF. Doing this depends on whether you use CXF in a Java EE application, OSGi Blueprint, Spring Boot, or standalone.

ADDING SWAGGER TO A CXF APPLICATION

To use Swagger with CXF, you need to create a `Swagger2Feature` and configure it. This can be done in a JAX-RS application class, as shown in the following listing.

Listing 10.17 Using a JAX-RS application to set up a REST

service with Swagger

```
@ApplicationPath("/")
public class RestOrderApplication extends Application
{   ①
```

①

JAX-RS Application class

```
    private final RestOrderService orderService;

    public RestOrderApplication(RestOrderService
orderService) {
        this.orderService = orderService;
    }

    @Override
    public Set<Object> getSingletons() {
        Swagger2Feature swagger = new
Swagger2Feature();  ②
```

②

Configures Swagger feature in CXF

```
        swagger.setBasePath("/");
        swagger.setHost("localhost:9000");
        swagger.setTitle("Order Service");
        swagger.setDescription("Rider Auto Parts Order
Service");  ②
        swagger.setVersion("2.0.0");
        swagger.setContact("rider@autoparts.com");
        swagger.setPrettyPrint(true);

        Set<Object> answer = new HashSet<>();  ③
```

③

Provides a set of features to use with CXF

```
        answer.add(orderService);  ③
        answer.add(new JacksonJsonProvider());  ③
        answer.add(swagger);  ③
```

```
        answer.add(new LoggingFeature());      ③
    return answer;
}
}
```

JAX-RS allows you to set up your JAX-RS RESTful services using your own class implementation that must extend `javax.ws.rs.core.Application`. You can see this in [listing 10.17](#), where you create the `RestOrderApplication` class ①. In the `getSingletons` method, you can set up the services that the RESTful application should use. To use Swagger with CXF, you need to create and configure an instance of `Swagger2Feature` ②. Then you configure the Rider Auto Parts `orderService` implementation, Jackson for JSON support, and Swagger, and then you enable logging by using the `LoggingFeature` ③.

To run this application standalone, you need to set up Jetty as an embedded HTTP server. And then you add CXF as a servlet to the HTTP server. The following listing shows how this can be done.

[Listing 10.18](#) Running Jetty with CXF JAX-RS and Swagger

```
public class RestOrderServer {

    public static void main(String[] args) throws Exception {
        DummyOrderService dummy = new DummyOrderService();
        dummy.setupDummyOrders();

        RestOrderService orderService = new RestOrderService();
        orderService.setOrderService(dummy);      ①
    }
}
```

①

Sets up the REST Order Service using dummy implementation

```
    RestOrderApplication app = new
    RestOrderApplication(orderService);      ②
}
```

②

Creates the JAX-RS application instance

```
Servlet servlet = new  
CXFNonSpringJaxrssServlet(app); ③
```

③

Uses CXF JAX-RS servlet without Spring

```
ServletHolder holder = new ServletHolder(servlet);  
holder.setName("rider");  
holder.setForcedPath("/");  
ServletContextHandler context = new  
ServletContextHandler();  
context.addServlet(holder, "/*");  
  
Server server = new Server(9000); ④
```

④

Sets up and starts Jetty embedded server on port 9000

```
server.setHandler(context);  
server.start();  
  
// keep that keeps the JVM running omitted  
}  
}
```

The example is run standalone, using a plain main class. The example uses a dummy order service **①**. An instance of the JAX-RS application is created **②**, which is then configured as a CXF JAX-RS servlet **③**. And the servlet is then added to an embedded Jetty server **④**, which is started.

The book's source code contains this example in the chapter10/cxf-swagger directory; you can try the example using the following Maven goal:

```
mvn compile exec:java
```

You can then access the Swagger documentation from the following URL:

```
http://localhost:9000/swagger.json
```

The returned Swagger API is verbose. [Figure 10.4](#) shows the API of the Get Order operation.

```
/orders/{id}: {
  - get: {
    - tags: [
      "orders"
    ],
    summary: "Get order",
    description: "",
    operationId: "getOrder",
    - consumes: [
      "application/json",
      "application/xml"
    ],
    - produces: [
      "application/json",
      "application/xml"
    ],
    - parameters: [
      - {
        name: "id",
        in: "path",
        description: "The id of the order",
        required: true,
        type: "integer",
        format: "int32"
      }
    ],
    - responses: {
      - 200: {
        description: "The found order",
        - schema: {
          $ref: "#/definitions/order"
        }
      },
      - 404: {
        description: "Cannot find order with the id",
        - schema: {
          type: "string"
        }
      }
    }
  },
}
```

Figure 10.4 Swagger API of the Get Order operation. Notice that the operation is detailed with a summary including the formats it consumes and produces, details of the input parameter, and the responses.

Even for this simple example with only four operations, the Swagger API can be verbose and detailed. We encourage you to try this example on your own and view the Swagger API from your web browser.

TIP You can install a JSON plugin to the browser that formats and outputs JSON in a more human-readable format. This example uses the JSONView extension in the Google Chrome browser, shown in [figure 10.4](#).

If you're using the Rest DSL with Camel, documenting your REST services becomes much easier.

10.3.2 USING SWAGGER WITH REST DSL

The Rest DSL comes with Swagger integration out of the box. You need to do two things to enable Swagger with Rest DSL:

- Add camel-swagger-java as a dependency
- Turn on Swagger in the Rest DSL configuration

To add camel-swagger-java as a dependency, you can add the following to your Maven pom.xml file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-swagger-java</artifactId>
    <version>2.20.1</version>
</dependency>
```

To turn on Swagger, you need to configure which context-path to use for the Swagger API service. This is done in the REST configuration using `apiContextPath`, as shown in Java DSL:

```
restConfiguration()
    .component("undertow").port(8080)
    .bindingMode(RestBindingMode.json_xml)
```

```
.dataFormatProperty("prettyPrint", "true")
.apiContextPath("api-doc");
```

And in XML DSL:

```
<restConfiguration component="undertow" port="8080"
                    bindingMode="json_xml"
                    apiContextPath="api-doc">
    <dataFormatProperty key="prettyPrint" value="true"/>
</restConfiguration>
```

The book's source code contains this example in the chapter10/undertow-swagger directory; you can run this example using the following Maven goal:

```
mvn compile exec:java
```

The Swagger API is available at <http://localhost:8080/api-doc>.

TIP You can specify whether the Swagger API should provide output in JSON or YAML. Use <http://localhost:8080/api-doc/swagger.json> for JSON and <http://localhost:8080/api-doc/swagger.yaml> for YAML.

How does this work?

HOW THE REST DSL SWAGGER WORKS

The Swagger API in the Rest DSL works in the same way as any of the other REST services defined in the Rest DSL. When `apiContextPath` is configured, the Swagger API is turned on. Under the hood, Camel creates a route that routes from `apiContextPath` to a rest-api endpoint. In Camel route lingo, this would be as follows:

```
from("undertow:http://localhost:8080/api-doc")
    .to("rest-api://api-doc");
```

The `rest-api` is a component from `camel-core` that seamlessly integrates with Swagger in the `camel-swagger-java` module.

When Camel starts up, the rest-api endpoint will discover that camel-swagger-java is available on the Java classpath and enable Swagger integration. What happens next is that the model of the REST services from the Rest DSL is parsed by a Swagger model reader from the camel-swagger-java module. The reader then transforms the Rest DSL into a Swagger API model. After a request to the Swagger API service, the Swagger API model is generated and transformed into the desired output format (either JSON or YAML) and returned.

All this requires no usage of the Swagger annotations in your source code. But they can be used to document your model objects. Besides using the Swagger annotations, you document your services directly by using the Rest DSL.

10.3.3 DOCUMENTING REST DSL SERVICES

In the introduction of section 10.3, we discussed the benefits of documenting your REST services in such a way that both humans and computers can understand all the service capabilities, without having to look up information from external resources such as offsite documentation or source code.

HATEOAS

Hypermedia as the Engine of Application State (HATEOAS) is an extension to RESTful principles that the services should be hypermedia driven. This principle goes the extra mile: a client needs no prior knowledge about how to interact with a service, besides the basic principles of hypermedia. An analogy is humans interacting with any website by using a web browser and following hyperlinks. HATEOAS uses the same principles: links are returned in response to clients, which can follow these links, and so on. The Rest DSL doesn't support HATEOAS.

The Rest DSL allows you to document the REST services directly in the DSL to include operations, parameters, response codes and types, and the like.

The following listing shows how the Rider Auto Parts order service application has been documented.

Listing 10.19 Document services using Rest DSL

```
rest("/orders")
    .description("Order services") ①
```

①

Description of the REST service and operations

```
    .get("{id}").outType(Order.class)
        .description("Get order by id") ①
        .param().name("id").description("The order
id").endParam() ②
```

②

Input parameter documentation

```
    .to("bean:orderService?method=getOrder(${header.id})")

    .post().type(Order.class).outType(String.class)
        .description("Create a new order") ①
```

①

Description of the REST service and operations

```
    .responseMessage() ③
```

③

Response message documentation

```
        .code(200).message("The created order id")
        .endResponseMessage()
        .to("bean:orderService?method=createOrder")
```

```
.put().type(Order.class)
.description("The order to update") ①
```

①

Description of the REST service and operations

```
.to("bean:orderService?method=updateOrder")

.delete("{id}")
.description("The order to cancel") ①
.param().name("id").description("The order
id").endParam() ②
```

②

Input parameter documentation

```
.to("bean:orderService?
method=cancelOrder(${header.id}))";
```

You use `description`, `param`, and `responseMessage` to document your Rest DSLs. `description` ① is used to provide a summary of the REST service and operations. `param` ② is used for documenting input parameters. And `responseMessage` ③ is used to document responses.

Two services have input parameters ②:

```
.param().name("id").description("The order id").endParam()
```

The parameter is named `id`, and it's the order ID, according to the description. The type of the parameter is a path parameter (default). A path parameter maps to a segment in the context-path, such as `http://localhost:8080/orders/{id}`, where `{id}` maps to the path parameter named `id` ②.

This book's accompanying source code contains this example using Java DSL in the `chapter10/undertow-swagger` directory. You can try this example using the following:

```
mvn compile exec:java
```

At startup, the example outputs the HTTP URLs you can use to call the services and access the API documentation. The API documentation is available in both JSON and YAML format.

The example is also provided using XML DSL in the chapter10/servlet-swagger-xml directory. This example is deployable as a web application, so you can deploy the .war artifact that's built into Apache Tomcat or WildFly servers. You can also run the example directly from Maven using the following:

```
mvn compile jetty:run
```

You also need to document what input and output your REST service accepts and returns.

10.3.4 DOCUMENTING INPUT, OUTPUT, AND ERROR CODES

This section covers configuring and documenting what input data and parameters your REST service accepts. Then we move on to documenting what your REST services can send as output responses. And we end the section covering document failures.

A RESTful service can have five types of input parameters in different forms and shapes:

- *body*—The HTTP body
- *formData*—Form data in the HTTP body
- *header*—An HTTP header
- *path*—A context-path segment
- *query*—An HTTP query parameter

The most common parameter types in use are body, path, and query. The previous examples covered so far in this chapter all used two parameter types: body and path.

THE INPUT PATH PARAMETER TYPE

The following snippet is the service used to get an order by providing an order ID:

```
.get("{id}").outType(Order.class)
    .to("bean:orderService?method=getOrder(${header.id})")
```

The path parameter type is in use only when the service uses { } in the URL. As you can see from the snippet, the URL is configured as {id} that will automatically let the Rest DSL define API documentation for this parameter.

You can explicitly declare the parameter type in the Rest DSL and configure additional details, as shown here:

```
.param().name("id").description("The order id").endParam()
```

You can also specify the expected data type. For example, you can document that the parameter is an integer value:

```
.param().name("id").dataType("int").description("The order
id").endParam()
```

And in XML DSL:

```
<param name="id" type="path" dataType="int"
description="The order id"/>
```

You want to do this only if you need to document the type with a human description or specify the data type. By default, the data type is string.

THE INPUT BODY PARAMETER TYPE

The body parameter type is used in the following snippet:

```
.put().type(Order.class)
    .to("bean:orderService?method=updateOrder")
```

It may not be obvious at first sight. When you use type(Order.class), the Rest DSL automatically defines that as a body parameter type. Because the type is a POJO, the data type of the parameter is a schema type. In the generated Swagger API documentation, the schemas are listed at the bottom.

[Figure 10.5](#) shows how the Swagger API documentation maps the body parameter type to the order definition.



```
put: {
  - tags: [
    "orders"
  ],
  summary: "Service to update an existing order",
  operationId: "route3",
  - consumes: [
    "application/json"
  ],
  - produces: [
    "application/json"
  ],
  - parameters: [
    - {
      in: "body",
      name: "body",
      description: "",
      required: true,
      - schema: {
        $ref: "#/definitions/order"
      }
    }
  ],
}

definitions: {
  - order: {
    type: "object",
    - required: [
      "amount",
      "customerName",
      "id",
      "partName"
    ],
    - properties: {
      - id: {
        type: "integer",
        format: "int32",
        description: "The order id"
      },
      - partName: {
        type: "string",
        description: "The name of the item to order"
      },
      - amount: {
        type: "integer",
        format: "int32",
        description: "Number of items to order"
      },
      - customerName: {
        type: "string",
        description: "The name of the customer"
      }
    },
    description: "An order"
  }
}
```

Figure 10.5 On the left, the `PUT` operation with a body input parameter refers to a schema definition/order. The schema is defined at the bottom of the Swagger API documentation, shown on the right.

OTHER INPUT PARAMETER TYPES

All the parameter types are configured in the same style; for example, to specify a query parameter named `priority`, you do this in Java DSL:

```
.param().name("priority").type(RestParamType.query)
.description("If the matter is urgent").endParam()
```

And in XML DSL:

```
<param name="priority" type="query" description="If the matter is urgent"/>
```

So far, we've covered the input parameter types. But there are also outgoing types.

OUTPUT TYPES

The response types are used for documenting what the RESTful services are returning. There are two kind of response types:

- *Body*—The HTTP body
- *Header*—An HTTP header

Let's take a look at how to use those.

THE BODY OUTPUT TYPE

The response body parameter is used in the following snippet:

```
.get("{id}").outType(Order.class)
    .param().name("id").description("The order
id").endParam()
    .to("bean:orderService?method=getOrder(${header.id})")
```

Here you're using `outType(Order.class)` that causes the Rest DSL to automatically define the response message as an `Order` type. Because the out type is a POJO, the data type of the response is a schema. [Figure 10.6](#) shows the generated Swagger API documentation for this REST service and how it refers to the order schema.

```

get: {
  - tags: [
    "orders"
  ],
  summary: "Service to get details of an existing order",
  operationId: "routet",
  - consumes: [
    "application/json"
  ],
  - produces: [
    "application/json"
  ],
  - parameters: [
    - {
      name: "id",
      in: "path",
      description: "The order id",
      required: true,
      type: "string"
    }
  ],
  - responses: {
    - 200: {
      description: "The order with the given id",
      - schema: {
        $ref: "#/definitions/order"
      }
    },
    - 404: {
      description: "Order not found"
    },
    - 500: {
      description: "Server error"
    }
  }
}

definitions: {
  - order: {
    type: "object",
    - required: [
      "amount",
      "customerName",
      "id",
      "partName"
    ],
    - properties: {
      - id: {
        type: "integer",
        format: "int32",
        description: "The order id"
      },
      - partName: {
        type: "string",
        description: "The name of the item to order"
      },
      - amount: {
        type: "integer",
        format: "int32",
        description: "Number of items to order"
      },
      - customerName: {
        type: "string",
        description: "The name of the customer"
      }
    },
    description: "An order"
  }
}

```

Figure 10.6 On the left is the `GET` operation with a successful (code 200) response body that refers to a schema definition/order. The schema is defined at the bottom of the Swagger API documentation, shown on the right.

You can also explicitly define response messages using a description understood by humans, shown highlighted here:

```

.get("{id}").outType(Order.class)
  .description("Service to get details of an existing
order")
  .param().name("id").description("The order
id").endParam()
  .responseMessage()
    .code(200).message("The order with the given id")
  .endResponseMessage()
  .to("bean:orderService?method=getOrder(${header.id})")

```

And in XML DSL:

```

<get uri="/{id}" outType="camelaction.Order">
  <description>Service to get details of an existing
order</description>
  <param name="id" type="path" description="The order id"/>
  <responseMessage code="200" message="The order with the
given id"/>
  <to uri="bean:orderService?

```

```
method=getOrder(${header.id})"/>
</get>
```

The code attribute refers to the HTTP status code; 200 is a successful call. In a short while, we'll show you how to document failures.

When a REST service returns a response, you most often use the HTTP body for the response, but you can use HTTP headers also.

THE HEADER OUTPUT TYPE

You can include HTTP headers in the response. If you do, you can document these headers in the Rest DSL, as highlighted here:

```
.get("{id}").outType(Order.class)
    .description("Service to get details of an existing
order")
    .param().name("id").description("The order
id").endParam()
    .responseMessage()
        .code(200).message("The order with the given id")
        .header("priority").dataType("boolean")
            .description("Priority order")
        .endHeader()
    .endResponseMessage()
.to("bean:orderService?method=getOrder(${header.id})")
```

And in XML DSL:

```
<get uri="/{id}" outType="camelaction.Order">
    <description>Service to get details of an existing
order</description>
    <param name="id" type="path" description="The order id"/>
    <responseMessage code="200" message="The order with the
given id">
        <header name="priority" dataType="boolean"
            description="Priority order"/>
    </responseMessage>
    <to uri="bean:orderService?
method=getOrder(${header.id})"/>
</get>
```

You can also document which failure codes a REST service can return.

DOCUMENTING ERROR CODES

The Rider Auto Parts order service application can return responses to clients using four status codes, listed in table [10.11](#).

Table 10.11 The four HTTP status codes the Rider Auto Parts order application can return

C o d e	Exception	Description
2 00		The request was processed successfully.
4 00	orderInvali lidExcept ion	A client attempted to create or update an order with invalid input data. Therefore, the client error code 400 is returned to indicate invalid input.
4 04	orderNotF oundExcep tion	A client attempted to get the status of a nonexistent order. Therefore, response code 404 is returned to indicate no data.
5 00	Exception	There was a server-side error processing the request. Therefore, a generic status code 500 is returned.

All four status codes are documented in the same way in the Rest DSL. The following snippets show two of the four REST services in the Rider Auto Parts application in Java and XML:

```
.get("{id}").outType(Order.class)
    .description("Service to get details of an existing
order")
    .param().name("id").description("The order
id").endParam()
    .responseMessage()
        .code(200).message("The order with the given
id").endResponseMessage()
    .responseMessage()
        .code(404).message("Order not
found").endResponseMessage()
    .responseMessage()
```

```
.code(500).message("Server error").endResponseMessage()
.to("bean:orderService?method=getOrder(${header.id})")
```

And the second service in XML DSL:

```
<post type="camelaction.Order" outType="String">
  <description>Service to submit a new order</description>
  <responseMessage code="200" message="The created order
id"/>
  <responseMessage code="400" message="Invalid input
data"/>
  <responseMessage code="500" message="Server error"/>
  <to uri="bean:orderService?method=createOrder"/>
</post>
```

How do you handle those thrown exceptions in the Rest DSL? Well, that's done using regular Camel routes in which you use Camel's error handler, such as `onException`. The following snippet shows how to handle the generic exception and transform that into an HTTP Status 500 error message:

```
onException(Exception.class)
  .handled(true)
  .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(500))
  .setBody(simple("${exception.message}\n"));
```

And in XML DSL:

```
<onException>
  <exception>java.lang.Exception</exception>
  <handled><constant>true</constant></handled>
  <setHeader headerName="Exchange.HTTP_RESPONSE_CODE">
    <constant>500</constant>
  </setHeader>
  <setBody>
    <simple>${exception.message}\n</simple>
  </setBody>
</onException>
```

TIP Don't worry if you don't completely understand all details of `onException`. Those details are extensively covered in chapter 11.

We encourage you to try the example that accompanies this book. You can find the Java-based example in chapter10/undertow-swagger, and the XML-based example in chapter10/servlet-swagger-xml. Both examples have a readme file with further instructions.

When using API documentation, you may need to use various configuration options.

10.3.5 CONFIGURING API DOCUMENTATION

The Rest DSL provides several configuration options for API documentation, as listed in table [10.12](#).

Table 10.12 API documentation options for Rest DSL

Option	Default	Description
api-component	swagger	Indicates the name of the Camel component to use as the REST API (such as swagger).
api-context-path		Configures a base context-path the REST API server will use. Important: you must specify a value for this option to enable API documentation.
api-c		Specifies the name of the route that the REST API server creates for hosting the API documentation.

o n t e x t R o u t e Id		
a p i c o n t e x t I d P a t t e rn	a p i c o n t e x t I d P a t t e rn	Sets a pattern to expose REST APIs only from REST services that are hosted within CamelContexts matching the pattern. You use this when running multiple Camel REST applications in the same JVM and you want to filter which camelContexts are exposed as REST APIs. The pattern name refers to the camelContext name, to match on the current CamelContext only.
a p i c o n t e x t L i s t i ng	f a l s e	Sets whether a listing of all available camelContexts with REST services in the JVM is enabled. If enabled, you can discover these contexts. If false, only the current CamelContext is in use.
e n a b l e C O RS	f a l s e	Indicates whether to enable CORS headers in the HTTP responses.

You'll often need to use only the `apiContextPath` option, as it's the one that enables API documentation. You won't use the first option, because currently Swagger is the only library integrated with Camel to service RESTful API documentation.

The third, fourth, and fifth options are all related to filtering out which `camelContexts` and routes to include in the API documentation. The last option is for enabling CORS, which we cover at the end of this section.

ENABLING API DOCUMENTATION

To use the Swagger API, you need to enable it in the Rest DSL configuration. To enable the Swagger API, you configure a context-path, which is used as the base path to service the API documentation.

The following snippet highlights how to configure this in Java:

```
restConfiguration()
    .component("undertow").port(8080)
    .bindingMode(RestBindingMode.json_xml)
    .dataFormatProperty("prettyPrint", "true")
    .apiContextPath("api-doc");
```

And in XML DSL:

```
<restConfiguration component="undertow"
bindingMode="json_xml"
    port="8080" apiContextPath="api-docs">
    <dataFormatProperty key="prettyPrint" value="true"/>
</restConfiguration>
```

When using Swagger API documentation, you'll likely want to provide a set of general information, such as version and contact information. The information is configured using `apiProperty`:

```
restConfiguration()
    .component("undertow").port(8080)
    .bindingMode(RestBindingMode.json_xml)
    .dataFormatProperty("prettyPrint", "true")
    .apiContextPath("api-doc")
    .apiProperty("api.version", "2.0.0")
    .apiProperty("api.title", "Rider Auto Parts Order")
```

```
Services")
    .apiProperty("api.description",
        "Order Service to submit orders and query status")
    .apiProperty("api.contact.name", "Rider Auto Parts");
```

And in XML DSL:

```
<restConfiguration component="undertow"
bindingMode="json_xml"
    port="8080" apiContextPath="api-docs">
    <dataFormatProperty key="prettyPrint" value="true"/>
    <apiProperty key="base.path" value="rest"/>
    <apiProperty key="api.version" value="2.0.0"/>
    <apiProperty key="api.title" value="Rider Auto Parts
Order Services"/>
    <apiProperty key="api.description"
        value="Order Service to submit orders and query
status"/>
    <apiProperty key="api.contact.name" value="Rider Auto
Parts"/>
</restConfiguration>
```

The possible values for the keys in the `apiProperty` are the `info` object from the Swagger API specification. You can find more details at <http://swagger.io/specification/#infoObject>.

FILTERING CAMELCONTEXT AND ROUTES IN API DOCUMENTATION

You may have some REST services that you don't want to be included in the public Swagger API documentation. The easiest way to disable a REST service is to set the `apiDocs` attribute to `false`:

```
.get("/ping").apiDocs(false)
    .to("direct:ping")
```

And in XML DSL:

```
<get uri="/ping" apiDocs="false">
    <to uri="direct:ping"/>
</get>
```

The other options (`apiContextListing`, `apiContextIdPattern`, and `apiContextRouteId`) are all options you may find usable only

when you run multiple Camel applications in the same JVM, such as from an application server. The idea is that you can enable API documentation in a single application that can discover and service API documentation for all the Camel applications running in the same JVM. This allows you to have a single endpoint as the entry to access API documentation for all the services running in the same JVM.

If you enable `apiContextListing`, the root path returns all the IDs of the Camel applications that have RESTful services.

The example in `chapter10/servlet-swagger-xml` has enabled the API context listing, which you can access using the following URL:

```
$ curl http://localhost:8080/chapter10-servlet-swagger-xml/rest/api-doc
[
  {"name": "camel-1"}
]
```

As you can see from the preceding output, only one Camel application in the JVM has RESTful services. The name is `camel-1`, which means the API documentation from within that Camel application is available at the following URL:

```
$ curl http://localhost:8080/chapter10-servlet-swagger-xml
  /rest/api-doc/camel-1
{
  "swagger" : "2.0",
  "info" : {
    "description" : "Order Service to submit orders and
query status",
    "version" : "2.0.0",
    "title" : "Rider Auto Parts Order Services",
    "contact" : {
      "name" : "Rider Auto Parts"
    }
  }
  ...
}
```

Swagger lets you visualize the API documentation via a web browser. The web browser requires online access to the Swagger API documentation. Because the web browser is separated from

the context-path where the Swagger API documentation resides, you need to enable CORS.

10.3.6 USING CORS AND THE SWAGGER WEB CONSOLE

Over the years, web browsers have become more secure and can't access resources outside the current domain. If a user visits the web page <http://rider.com>, that website can freely load resources from the rider.com domain. But if the web page attempts to load resources from outside that domain, the web browser would disallow that.

But it's not that simple, as some resources are always allowed (such as CSS styles, images, and scripts), but advanced requests (such as POST, PUT, and DELETE) aren't allowed. *Cross-Origin Resource Sharing (CORS)* is a way of allowing clients and servers to negotiate whether the client can access those advanced resources. The negotiation happens using a set of HTTP headers in which the client specifies the resource it wants access to, and the server then responds with a set of HTTP headers that tells the client what's allowed.

Our web browsers of today are all CORS compliant and perform these actions out of the box.

Why do we need CORS, you may ask? You may need CORS, for example, if you build web applications using JavaScript technology that calls RESTful services on remote servers.

It's easy to enable CORS when using the Rest DSL, which is done in the REST configuration:

```
restConfiguration()
    .component("undertow").port(8080)
    .bindingMode(RestBindingMode.json_xml)
    .dataFormatProperty("prettyPrint", "true")
    .enableCORS(true)
    .apiContextPath("api-doc")
```

And in XML:

```

<restConfiguration component="servlet" bindingMode="json"
    contextPath="chapter10-swagger-ui/rest"
    port="8080"
        apiContextPath="api-doc"
    apiContextListing="true"
        enableCORS="true">

```

When CORS is enabled, the CORS headers in table [10.13](#) are in use.

Table 10.13 Default CORS headers in use

HTTP header	Value
Access-Control-Allow-Origin	*
Access-Control-Allow-Methods	GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH
Access-Control-Allow-Headers	Origin, Accept, X-Requested-With, Content-Type, Access-Control-Request-Method, Access-Control-Request-Headers
Access-Control-Max-Age	3600

If the default value isn't what you need, you can customize the CORS headers in the REST configuration:

```

restConfiguration()
    .component("undertow").port(8080)
    .bindingMode(RestBindingMode.json_xml)
    .dataFormatProperty("prettyPrint", "true")
    .enableCORS(true)
    .apiContextPath("api-doc")
    .apiProperty("api.version", "2.0.0")
    .apiProperty("api.title", "Rider Auto Parts Order
Services")
    .apiProperty("api.description",
        "Order Service to submit orders and query status")
    .apiProperty("api.contact.name", "Rider Auto Parts")
    .corsHeaderProperty("Access-Control-Allow-Origin",
"rider.com")
    .corsHeaderProperty("Access-Control-Max-Age", "300");

```

And in XML DSL:

```
<restConfiguration component="servlet" bindingMode="json"
    contextPath="chapter10-swagger-ui/rest"
    port="8080"
        apiContextPath="api-doc"
    apiContextListing="true"
        enableCORS="true">
    <dataFormatProperty key="prettyPrint" value="true"/>
    <apiProperty key="base.path" value="rest"/>
    <apiProperty key="api.version" value="2.0.0"/>
    <apiProperty key="api.title" value="Rider Auto Parts
Order Services"/>
    <apiProperty key="api.description"
        value="Order Service to submit orders and query
status"/>
    <apiProperty key="api.contact.name" value="Rider Auto
Parts"/>
    <corsHeaders key="Access-Control-Allow-Origin"
value="rider.com"/>
    <corsHeaders key="Access-Control-Max-Age" value="300"/>
</restConfiguration>
```

Using this configuration, only clients from the domain rider.com are allowed to access the RESTful services by using CORS. The Max-Age option is set to 5 minutes; the client is allowed to cache a preflight request to a resource for that amount of time before the client must renew and call a new preflight request.

Let's put CORS to use and use the Swagger UI web application.

EMBEDDING SWAGGER UI WEB CONSOLE

The Swagger API documentation can be visualized using the Swagger UI. This web console also allows you to try calling the REST services, making it a great tool for developers. The console consists of HTML pages and scripts and has no server-side dependencies. Therefore, you can more easily host or embed the console on application servers or locally.

The book's source code contains an example in which the web console is embedded together in a Camel application. The example is located in the chapter10/swagger-ui directory; you

can try the example using the following Maven goals:

```
mvn compile jetty:run-war
```

Then from a web browser, open the following URL:

```
http://localhost:8080/chapter10-swagger-ui/?url=rest/api-doc/camel-1/swagger.json
```

This loads the Swagger API documentation from the REST services and presents the API documentation beautifully in the web console. You can then click the Expand Operations button and see all four services. Then you can open the GET operation, type in 1 as the order ID, and click the Execute button.

Now let's try to browse the Swagger API documentation from a remote service. We have another example in the source code, in the chapter10/spark-rest-ping directory. Now start this example by using the following:

```
mvn compile exec:java
```

This starts a ping REST service that you can access with this:

```
http://localhost:9090/ping
```

And the Swagger API for the ping service is available here:

```
http://localhost:9090/api-doc
```

You can then, from the Swagger UI, view the ping service by typing at the top of the web page the URL for the ping API documentation; then click the Explore button, as shown in [figure 10.7](#).

The screenshot shows the Swagger UI interface for a REST API. At the top, there's a green header bar with the 'swagger' logo, a URL input field containing 'http://localhost:9090/api-doc/swagger.json', and a 'Explore' button. Below the header, there's a dropdown menu for 'Schemes' set to 'HTTP'. The main content area has a title 'ping ▾'. Underneath, a 'GET /ping' operation is listed with a 'Parameters' section showing 'No parameters'. There are 'Execute' and 'Clear' buttons. In the 'Responses' section, the 'Response content type' is set to 'application/json'. Below this, a 'Curl' section contains the command: 'curl -X GET "http://0.0.0.0:9090/ping" -H "accept: application/json"'. The 'Server response' section shows a 200 status code with a 'Response body' containing the JSON object: { "reply": "pong" }.

Figure 10.7 Swagger UI browsing the remote ping REST API documentation. At the top of the window, you type the URL to the Swagger API you want to browse and then click the Explore button. Then you can Expand Operations and try calling the services. In the figure, we've called the ping service and received a pong reply in the Response Body section.

The Swagger UI is able to browse and call the remote REST services only because we've enabled CORS. You can view the CORS headers using curl or similar commands:

```
$ curl -i http://localhost:9090/ping
HTTP/1.1 200 OK
```

```
Date: Sat, 25 Mar 2017 16:57:42 GMT
Accept: /*
Access-Control-Allow-Headers: Origin, Accept, X-Requested-With, Content-Type, Access-Control-Request-Method, Access-Control-Request-Headers
Access-Control-Allow-Methods: GET, HEAD, POST, PUT, DELETE, TRACE,
                             OPTIONS, CONNECT, PATCH
Access-Control-Allow-Origin: *
Access-Control-Max-Age: 3600
breadcrumbId: ID-davsclaus-air-63483-1458925056040-0-1
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Server: Jetty(9.2.21.v20160210)

{ "reply": "pong" }
```

Whoa, that was a lot of content to cover about RESTful services! Who would've thought there's so much to learn and master when using such a popular and modern means of communication?

That's it folks. That's all we want to say about RESTful web services.

10.4 Summary and best practices

In this chapter, you looked at building RESTful services with Camel, using the wide range of choices Camel offers. In fact, you started without Camel, built a pure JAX-RS service using CXF, and then explored Camel in various stages. You went through all the other REST components at your disposal and reviewed a short summary of the pros and cons of each choice.

REST services is becoming a top choice for providing and consuming services. In light of this, Camel introduced a specialized Rest DSL that makes it easier for Camel developers to build REST services. This chapter covered all about the Rest DSL.

Swagger is becoming the de facto standard for an API documenting your REST services. You saw how to document your REST services in the Rest DSL, which then is directly

serviced as Swagger API documentation out of the box. To view Swagger API documentation, we showed you how to use the Swagger UI web console, which when CORS is enabled, allows you to browse and access remote REST services.

We've listed the takeaways for this chapter:

- *JAX-RS is a good standard*—The JAX-RS specification allows Java developers to code RESTful services in an annotation-driven style. The resource class allows the developer full control of handling the REST services. Apache CXF integrates the JAX-RS specification and is a good RESTful framework.
- *Java code or Camel route*—Apache Camel allows you to expose Camel routes as if they're RESTful services. You need to decide whether you want the full coding power that the JAX-RS resource class allows or to go straight to Camel routes. When you need both, you can use the hybrid mode with a JAX-RS resource class and call Camel routes by using `ProducerTemplate` (as shown in section 10.1.3).
- *JAX-RS or the Rest DSL*—The Rest DSL is a powerful DSL that allows new users of REST to quickly understand and develop REST services that integrate well with the philosophy and principles of Apache Camel. The Rest DSL has started to support calling REST services from Camel 2.19 onward. It's expected that this will become even easier and smarter in upcoming Camel releases.
- *Use the right REST components*—The Rest DSL allows you to use many Camel components to host the REST services that are easy to swap out. You can start using Jetty and later change the servlet if you run in a Java EE application server. You can also go minimal with Netty or Undertow. Or change to CXF or Restlet, which are first-class RESTful frameworks.
- *Document your REST services*—Your REST services can be documented using the Rest DSL and automatically provide API documentation using Swagger at runtime.

- *Govern and manage your APIs*—There was no room in this chapter to touch on this topic, but we want to share our views that API management becomes important when you have many APIs to integrate.

We've covered a lot of ground pertaining to RESTful web services. Now we'll take a leap into another world, one that's often tackled as an afterthought in integration projects: how to handle situations when things go wrong. We've devoted an entire chapter to Camel's extensive support for error handling.

Part 4

Going further with Camel

In complex enterprise systems, lots of things can go wrong. That's why Camel features an extensive set of error-handling abilities. In chapter 11 we'll discuss these in detail.

One concept you'll encounter in enterprise applications is transactions. In chapter 12, we'll explain how to use Spring's transaction framework to let Camel routes participate in transactions. In case of a lack of transactional support, you can use compensation and the idempotency patterns, which are covered as well.

In chapter 13 we'll discuss the important, sometimes complex topic of concurrency. Understanding how to configure and tune threading in Camel is a must for performance-centric projects. We'll also cover how you can improve the scalability of your Camel applications.

Building secured applications isn't easy, and we devote chapter 14 to walking you through the various ways of building security into your Camel applications.

11

Error handling

This chapter covers

- Understanding error handling
- Knowing where and when Camel's error handling applies
- Using the various error handlers in Camel
- Using redelivery policies
- Handling exceptions with `onException`
- Reusing error handlers in all your routes
- Performing fine-grained control of error handling

You've now reached the halfway mark of this book, and we've covered a lot of ground. But what you're about to embark on in this chapter is one of the toughest topics in integration: how to deal with the unexpected.

Writing applications that integrate disparate systems is a challenge when it comes to handling unexpected events. In a single system that you fully control, you can handle these events and recover. But systems that are integrated over the network have additional risks: the network connection could be broken, a remote system might not respond in a timely manner, or it might even fail for no apparent reason. Even on your local server, unexpected events can occur, such as the server's disk filling up

or the server running out of memory. Regardless of which errors occur, your application should be prepared to handle them.

In these situations, log files are often the only evidence of the unexpected event, so logging is important. Camel has extensive support for logging and for handling errors to ensure that your application can continue to operate.

In this chapter, you'll discover how flexible, deep, and comprehensive Camel's error handling is and how to tailor it to deal with most situations. We'll cover all the error handlers Camel provides out of the box, and when they're best used, so you can pick the ones suited to your applications. You'll also learn how to configure and master redelivery, so Camel can try to recover from particular errors. We'll also look at exception policies, which allow you to differentiate among errors and handle specific ones, and at how scopes can help you define general rules for implementing route-scoped error handling. Finally, we'll look at what Camel offers when you need fine-grained control over error handling, so that it reacts under only certain conditions.

This chapter covers a complicated topic that Camel users may have trouble learning. Therefore, we've taken the time and space to cover all aspects of this in detail, and as a result this chapter is long and not a light read. We suggest you take a break halfway through; we'll tell you when.

11.1 Understanding error handling

Before jumping into the world of error handling with Camel, you need to take a step back and look at errors more generally. You also need to look at where and when error handling starts, because some prerequisites must happen beforehand.

11.1.1 RECOVERABLE AND IRRECOVERABLE ERRORS

When it comes to errors, you can divide them into two main

categories: recoverable and irrecoverable errors, as illustrated in [figure 11.1](#).

An *irrecoverable error* remains an error no matter how many times you try to perform the same action again. In the integration space, that could mean trying to access a database table that doesn't exist, which would cause the JDBC driver to throw `SQLException`.

A *recoverable error*, on the other hand, is a temporary error that might not cause a problem on the next attempt. A good example of such an error is a problem with the network connection resulting in `java.io.IOException`. On a subsequent attempt, the network issue could be resolved, and your application could continue to operate.

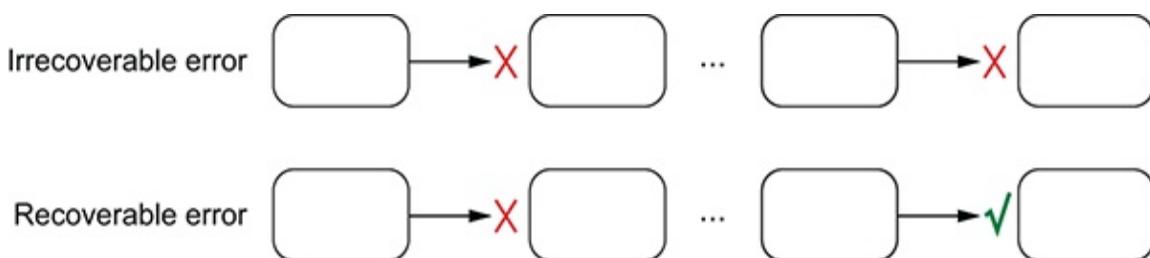


Figure 11.1 Errors can be categorized as either recoverable or irrecoverable. Irrecoverable errors continue to be errors on subsequent attempts; recoverable errors may be quickly resolved on their own.

In your daily life as a Java developer, you won't often encounter this division of errors into recoverable and irrecoverable. Generally, exception-handling code uses one of the two patterns illustrated in the following two code snippets.

The first snippet illustrates a common error-handling idiom: all kinds of exceptions are considered irrecoverable, and you give up immediately, throwing the exception back to the caller, often wrapped:

```
public void handleOrder(Order order) throws  
OrderFailedException {  
    try {  
        service.sendOrder(order);  
    } catch (Exception e) {  
        throw new OrderFailedException(e);  
    }  
}
```

```
    }
}
```

The next snippet improves on this situation by adding a bit of logic to handle redelivery attempts before eventually giving up:

```
public void handleOrder(Order order) throws
OrderFailedException {
    boolean done = false;
    int retries = 5;
    while (!done) {
        try {
            service.sendOrder(order);
            done = true;
        } catch (Exception e) {
            if (--retries == 0) {
                throw new OrderFailedException(e);
            }
        }
    }
}
```

Around the invocation of the service is the logic that attempts redelivery, in case an error occurs. After five attempts, it gives up and throws the exception.

What the preceding example lacks is logic to determine whether the error is recoverable or irrecoverable, and to react accordingly. In the recoverable case, you could try again, and in the irrecoverable case, you could give up immediately and rethrow the exception.

In Camel, a recoverable error is represented as a plain `Throwable` or `Exception` that can be set or accessed from `org.apache.camel.Exchange` by using one of the following methods:

```
void setException(Throwable cause);
```

or

```
Exception getException();
<T> T getException(Class<T> type);
```

NOTE The `setException` method on `Exchange` accepts a `Throwable` type, whereas the `getException` method returns an `Exception` type. `getException` also doesn't return a `Throwable` type because of backward API compatibility. The second `getException` method accepts a class type that allows you to traverse the exception hierarchy to find a matching exception. Section 11.4 covers how this works.

An irrecoverable error is represented as a message with a fault flag that can be set or accessed from `org.apache.camel.Exchange`. For example, to set `Unknown customer` as a fault message, you would do the following:

```
Message msg = Exchange.getOut();
msg.setFault(true);
msg.setBody("Unknown customer");
```

The fault flag must be set using the `setFault(true)` method.

Why are the two types of errors represented differently? Two reasons. First, the Camel API was designed around the Java Business Integration (JBI) specification, which includes a fault message concept. Second, Camel has error handling built into its core, so whenever an exception is thrown back to Camel, it catches it and sets the thrown exception on the exchange as a recoverable error, as illustrated here:

```
try {
    processor.process(exchange);
} catch (Throwable e) {
    exchange.setException(e);
}
```

Using this pattern allows Camel to catch and handle all exceptions that are thrown. Camel's error handling can then determine how to deal with the errors: retry, propagate the error back to the caller, or do something else. End users of Camel can set irrecoverable errors as fault messages, and Camel can react accordingly and stop routing the message.

Fault or exception

In practice, you can represent Exception as a nonrecoverable error as well by using exception classes that by nature are nonrecoverable. For example, an `InvalidOrderIdException` exception represents a nonrecoverable error, as a given order ID is invalid. It's often more common with Camel to use this approach than to use fault messages. Fault messages are often only used with legacy components such as JBI (https://en.wikipedia.org/wiki/Java_Business_Integration) or SOAP web services.

Now that you've seen recoverable and irrecoverable errors in action, let's summarize how they're represented in Camel:

- Recoverable errors are represented as exceptions.
- Irrecoverable errors are represented as fault messages.

Next let's look at when and where Camel's error handling applies.

11.1.2 WHERE CAMEL'S ERROR HANDLING APPLIES

Camel's error handling doesn't apply everywhere. To understand why, take a look at [figure 11.2](#).

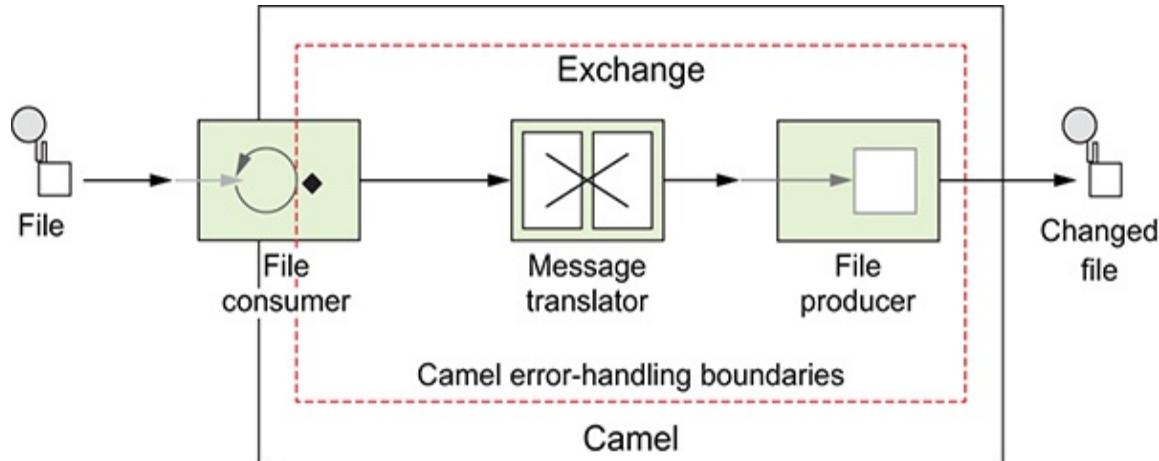


Figure 11.2 Camel’s error handling applies only within the lifecycle of an exchange. The error-handler boundaries are represented by the dashed square in the figure.

Figure 11.2 shows a simple route that translates files. You have a file consumer and producer as the input and output facilities, and in between is the Camel routing engine, which routes messages encompassed in an exchange. It’s during the lifecycle of this exchange that the Camel error handling applies. That leaves a little room on the input side where this error handling can’t operate—the file consumer must be able to successfully read the file, instantiate the exchange, and start the routing before the error handling can function. This applies to any kind of Camel consumer.

What happens if the file consumer can’t read the file? The answer is component specific, and each Camel component must deal with this in its own way. Some components will ignore and skip the message, others will retry a certain number of times, and others will gracefully recover. In the case of the file consumer, a `WARN` will be logged and the file will be skipped, and a new attempt to read the file will occur on the next polling.

But what if you want to handle this differently? What if instead of logging a `WARN`, you want to let the Camel error handler handle the exception, or what if the file keeps failing on the subsequent polls? These are all great questions that you’ll learn how to deal with when you have more experience with Camel’s error handler. You’ll come back to these questions and get answers in section

11.4.8. What you need to understand for now is that [figure 11.2](#) represents the default behavior for where Camel's error handler works.

That's enough background information. Let's dig into how error handling in Camel works. In the next section, you'll start by looking at the various error handlers Camel provides.

11.2 Using error handlers in Camel

In the last section, you learned that Camel regards all exceptions as recoverable and stores them on the exchange by using the `setException(Throwable cause)` method. This means error handlers in Camel will react only to exceptions set on the exchange. The rule of thumb is that error handlers in Camel trigger only when `exchange.getException() != null`.

Camel provides a range of error handlers. They're listed in [table 11.1](#).

Table 11.1 Error handlers provided in Camel

Error Handler	Description
DefaultErrorHandler	This is the default error handler that's automatically enabled, in case no other has been configured.
DeadLetterChannel	This error handler implements the Dead Letter Channel EIP.
TransactionErrorHandler	This is a transaction-aware error handler extending the default error handler. Transactions are covered in the chapter 12 and are only briefly touched on in this chapter.
NoErrorHandler	This handler is used to disable error handling altogether.
LoggingErrorHandler	This error handler just logs the exception. This error handler is deprecated, in favor of using the <code>DeadLetterChannel</code> error handler, using a log endpoint as the destination.

At first glance, having five error handlers may seem

overwhelming, but you'll learn that the default error handler is used in most cases.

The first three error handlers in table 11.1 all extend the `RedeliveryErrorHandler` class. That class contains the majority of the error-handling logic that the first three error handlers all use. The latter two error handlers have limited functionality and don't extend `RedeliveryErrorHandler`. We'll look at each of these error handlers in turn.

11.2.1 USING THE DEFAULT ERROR HANDLER

Camel is preconfigured to use `DefaultErrorHandler`, which covers most use-cases. To understand it, consider the following route:

```
from("direct:newOrder")
    .bean("orderService", "validate")
    .bean("orderService", "store");
```

The default error handler is preconfigured and doesn't need to be explicitly declared in the route. What happens if an exception is thrown from the `validate` method on the order service bean?

To answer this, you need to dive into Camel's inner processing, where the error handler lives. In every Camel route, a channel sits between each node in the route path, as illustrated in figure 11.3.

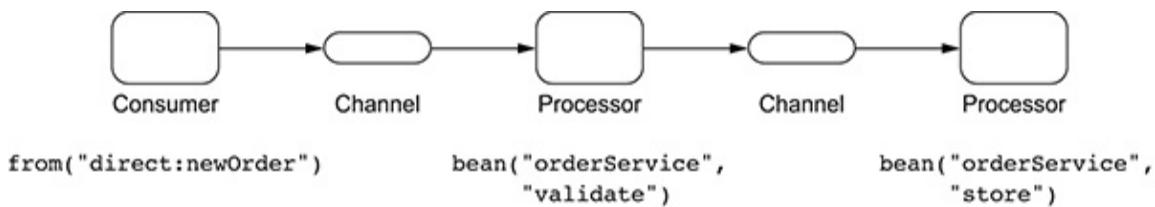


Figure 11.3 A detailed view of a route path, where channels act as controllers between the processors

A channel is between each node of the route path, which ensures that it can act as a controller to monitor and control the routing at runtime. This is the feature that allows Camel to enrich the route with error handling, message tracing, interceptors, and much more. For now, you just need to know that this is where the error handler lives.

Turning back to the example route, imagine that an exception was thrown from the order service bean during invocation of the validate method. In [figure 11.3](#), the processor would throw an exception, which would be propagated back to the previous channel, where the error handler would catch it. This gives Camel the chance to react accordingly. For example, Camel could try again (redeliver), or it could route the message to another route path (detour using exception policies), or it could give up and propagate the exception back to the caller. With the default settings, Camel will propagate the exception back to the caller.

The default error handler is configured with these settings:

- No redelivery occurs.
- Exceptions are propagated back to the caller.
- The stack trace of the exception is printed to the log.
- The routing history of the exchange is printed to the log.

These settings match what happens when you’re working with exceptions in Java, so Camel’s behavior won’t surprise Camel end users.

When an exception is thrown during routing, the default behavior is to log a route history that traces the steps back to where the exchange was routed by Camel. This allows you to quickly identify where in the route the exception occurred. In addition, Camel logs details from the current exchange, such as the message body and headers. Section 11.3.3 covers this in more detail.

Let’s continue with the next error handler, the dead letter channel.

11.2.2 THE DEAD LETTER CHANNEL ERROR HANDLER

The `DeadLetterChannel` error handler is similar to the default error handler except for the following differences:

- The dead letter channel is the only error handler that supports moving failed messages to a dedicated error queue, known as the *dead letter queue*.
- Unlike the default error handler, the dead letter channel will, by default, handle exceptions and move the failed messages to the dead letter queue.
- The dead letter channel is by default configured to not log any activity when it handles exceptions.
- The dead letter channel supports using the original input message when a message is moved to the dead letter queue.

Let's look at each of these in a bit more detail.

THE DEAD LETTER CHANNEL

`DeadLetterChannel` is an error handler that implements the principles of the Dead Letter Channel EIP. This pattern states that if a message can't be processed or delivered, it should be moved to a dead letter queue. Figure 11.4 illustrates this pattern.

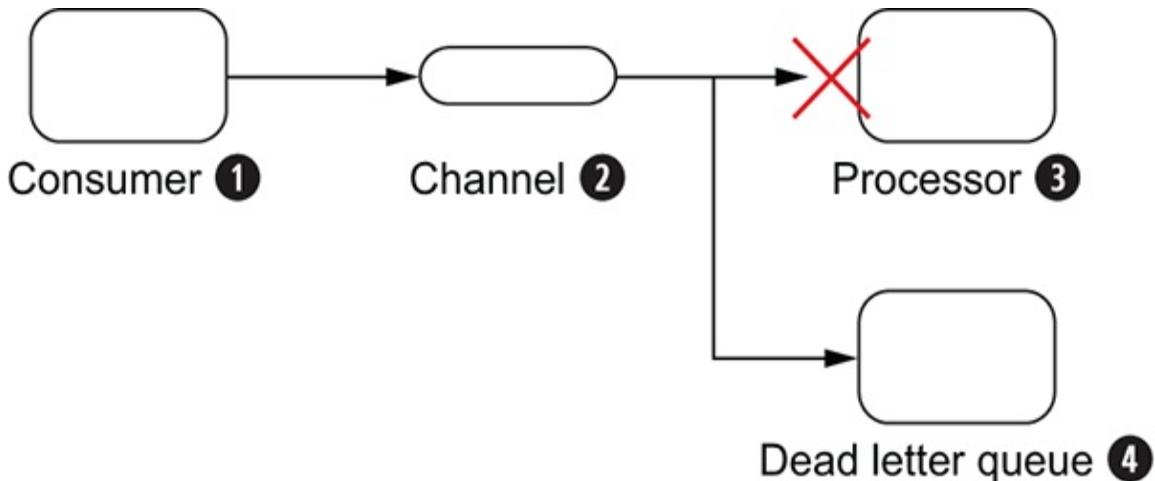


Figure 11.4 The Dead Letter Channel EIP moves failed messages to a dead letter queue.

As you can see, the consumer **1** consumes a new message that's supposed to be routed to the processor **3**. The channel **2** controls the routing between **1** and **3**, and if the message can't

be delivered to the processor ③, the channel invokes the dead letter channel error handler, which moves the message to the dead letter queue ④. This keeps the message safe and allows the application to continue operating.

This pattern is often used with messaging. Instead of allowing a failed message to block new messages from being picked up, the message is moved to a dead letter queue to get it out of the way.

The same idea applies to the dead letter channel error handler in Camel. This error handler has an associated dead letter queue, which is based on an endpoint, allowing you to use any Camel endpoint you choose. For example, you can use a database or a file, or just log the failed messages.

NOTE The situation gets a bit more complicated when using transactions together with a dead letter queue. Chapter 12 covers this in more detail.

When you choose to use the dead letter channel error handler, you must configure the dead letter queue as an endpoint so the handler knows where to move the failed messages. This is done a bit differently in the Java DSL and XML DSL. For example, here's how to log the message at `ERROR` level in Java DSL:

```
errorHandler(deadLetterChannel("log:dead?level=ERROR"));
```

When using Java DSL, the error handler is configured in the `RouteBuilder` classes, as shown in the following code:

```
public void configure() throws Exception {  
    errorHandler(deadLetterChannel("log:dead?  
level=ERROR"));      ①
```

①

Configures error handler for all routes in this `RouteBuilder` class

```
from("direct:newOrder") ②
```

②

Defines the Camel route(s)

```
.bean("orderService", "validate")
.bean("orderService", "store");
}
```

This configuration defines a `RouteBuilder`-scoped error handler **①** that applies to all the following routes **②** defined in the same class:

```
<errorHandler id="myErrorHandler" type="DeadLetterChannel"
    deadLetterUri="log:dead?level=ERROR"/>
```

Notice the difference between Java and XML DSL. In XML DSL, the `type` attribute is used to declare which error handler to use. In XML, the error handler must be configured with an ID, which would be required to enable the error handler on either the context or route scope level. The following listing is an equivalent of the prior Java DSL listing:

```
<camelContext errorHandlerRef="myErrorHandler">
```

```
    <errorHandler id="myErrorHandler"
        type="DeadLetterChannel" ①
```

①

Configures error handler

```
<route> ③ deadLetterUri="log:dead?level=ERROR"/>
```

③

Defines the Camel route(s)

```
<from uri="direct:newOrder"/>
<bean ref="orderService" method="validate"/>
<bean ref="orderService" method="store"/>
```

```
</route>
</camelContext>
```

Inside `<camelContext>`, you configure the error handler ❶. The error handlers in XML DSL can be used in two levels:

- Context-scoped level
- Route-scoped level

A route-scoped error handler takes precedence over a context-scoped error handler. In the preceding code, you have to configure the error handler as a context scope ❷ by referring to the error handler by its ID. The following routes ❸ then by default use the context-scoped error handler.

Now, let's look at how the dead letter channel error handler handles exceptions when it moves the message to the dead letter queue.

HANDLING EXCEPTIONS BY DEFAULT

By default, Camel handles exceptions by suppressing them; it removes the exceptions from the exchange and stores them as properties on the exchange. After a message has been moved to the dead letter queue, Camel stops routing the message, and the caller regards it as processed.

When a message is moved to the dead letter queue, you can obtain the exception from the exchange by using the `Exchange.EXCEPTION_CAUGHT` property:

```
Exception e =
exchange.getProperty(Exchange.EXCEPTION_CAUGHT,
Exception.class);
```

Now let's look at using the original message.

USING THE ORIGINAL MESSAGE WITH THE DEAD LETTER CHANNEL

Suppose you have a route in which the message goes through a series of processing steps, each altering a bit of the message

before it reaches its final destination, as in the following code:

```
errorHandler(deadLetterChannel("jms:queue:dead"));

from("jms:queue:inbox")
    .bean("orderService", "decrypt")
    .bean("orderService", "validate")
    .bean("orderService", "enrich")
    .to("jms:queue:order");
```

Now imagine that an exception occurs at the validate method, and the dead letter channel error handler moves the message to the dead letter queue. Suppose a new message arrives and an exception occurs at the enrich method, and this message is also moved to the same dead letter queue. If you want to retry those messages, can you just drop them into the inbox queue?

In theory, you could do that, but the messages that were moved to the dead letter queue no longer match the messages that originally arrived at the inbox queue—they were altered as the messages were routed. What you want instead is for the original message content to have been moved to the dead letter queue so that you have the original message to retry.

The `useOriginalMessage` option instructs Camel to use the original message when it moves messages to the dead letter queue. You configure the error handler to use the `useOriginalMessage` option as follows:

```
errorHandler(deadLetterChannel("jms:queue:dead")).useOriginalMessage();
```

In XML DSL, you'd do this:

```
<errorHandler id="myErrorHandler" type="DeadLetterChannel"
              deadLetterUri="jms:queue:dead"
              useOriginalMessage="true"/>
```

When the original message (or any other message, for that matter) is moved to a JMS dead letter queue, the message doesn't include any information about the cause of the error, such as the exception message or its stacktrace. The cause of the exception is stored as a property on the exchange, and exchange

properties aren't part of the JMS message that Camel sends. To include such information, you'd need to enrich the message with such details before being sent to the `jms:queue:dead` destination.

ENRICHING MESSAGE WITH CAUSE OF ERROR

To enrich the message with the cause of the error, you can use a Camel processor to implement the logic for enriching the message. And then you configure the error handler to use the processor. The following listing presents an example.

Listing 11.1 Using a processor to enrich the message before it's sent to the dead letter channel

```
public class FailureProcessor implements Processor { 1
```

1

Processor to enrich the message with details of why it failed

```
    public void process(Exchange exchange) throws Exception {  
        Exception e =  
exchange.getProperty(Exchange.EXCEPTION_CAUGHT,  
                           Exception.class);  
        String failure = "The message failed because " +  
e.getMessage();  
        exchange.getIn().setHeader("FailureMessage",  
failure); 2
```

2

Stores the failure reason as a header on the exchange

```
    }  
  
    errorHandler(deadLetterChannel("jms:queue:dead")  
        .useOriginalMessage().onPrepareFailure(new  
FailureProcessor()); 3
```

3

Configures the error handler to use the processor to prepare the failed exchange

To enrich the message, you can implement the details in `org.apache.camel.Processor` ①. Because you use a JMS message queue as the dead letter queue, you must store the failure reason on the exchange in the message payload; exchange properties aren't included in JMS messaging. Therefore, you use a message header ② to store the failure reason. You then need to configure the `DeadLetterChannel` error handler to use the processor before the exchange is sent to the dead letter queue ③.

Configuring the error handler by using XML DSL is slightly different, as you need to refer to `Processor` by using the `onPrepareFailureRef` attribute on the error handler:

```
<bean id="failureProcessor"
  class="org.camelaction.FailureProcessor"/>

<camelContext errorHandlerRef="myErrorHandler" . . .>
  <errorHandler id="myErrorHandler"
    type="DeadLetterChannel"
    deadLetterUri="jms:queue:dead"
    useOriginalMessage="true"
    onPrepareFailureRef="failureProcessor"/>
  . . .
</camelContext>
```

The book's source code contains this example in the `chapter11/enrich` directory. To try the example, use the following Maven goal for either the Java or XML version:

```
mvn test -Dtest=EnrichFailureProcessorTest
mvn test -Dtest=SpringEnrichFailureProcessorTest
```

Instead of using `Processor` to enrich the failed message, you can also use a Camel route, and let the dead letter endpoint call to the route by using the direct component. The previous example can be implemented using a route, as shown in the following listing.

[Listing 11.2](#) Using a route with a bean to enrich the message

before it's sent to the dead letter channel

```
public class FailureBean { ①
```

①

Bean to enrich the message with details of why it failed

```
    public void enrich(@Headers Map headers, Exception cause)  
    {  
        String failure = "The message failed because " +  
e.getMessage();  
        headers.put("FailureMessage", failure); ②
```

②

Stores the failure reason as a header on the message

```
} }
```

```
errorHandler(deadLetterChannel("direct:dead").useOriginalMe  
ssage()); ③
```

③

Configures the error handler to use a route to prepare the failed exchange

```
from("direct:dead") ④
```

④

The route that enriches the exchange by calling the bean and sends the message to the dead letter queue, which is mock:dead in this example

```
.bean(new FailureBean())  
.to("mock:dead"); ⑤
```

⑤

Using mock endpoint as the dead letter queue

You use a Java bean to enrich the message **1**. Using a Java bean allows you to use Camel’s parameter binding in the method signature. The first parameter is the message header, which is mapped using the `@Headers` annotation. The second parameter is the cause of the exception. You then construct the error message **2**, which you add to the headers. The error handler is configured to call the “`direct:dead`” endpoint **3**, which is a route that routes to the bean **4**. After the message has been enriched from the bean, the message is then routed to its final destination, which is the dead letter queue **5**.

The book’s source code contains this example in the `chapter11/enrich` directory. To try the example, use the following Maven goal for either the Java or XML version:

```
mvn test -Dtest=EnrichFailureBeanTest  
mvn test -Dtest=SpringEnrichFailureBeanTest
```

In this example, the route is just calling a bean, but you’re free to expand the route to do more work. There’s only one consideration—the fact that the route is triggered by the error handler. What if the route causes a new exception to happen? Would the error handler not react again, and call the `direct:dead` route with the new exception, and if the new exception also caused yet another new exception, don’t we have a potential endless loop with error handling? Yes, and that’s why Camel has a built-in fatal error handler that detects this and breaks out of this situation. Section 11.4.5 covers this in more detail.

Let’s move on to the transaction error handler.

11.2.3 THE TRANSACTION ERROR HANDLER

`TransactionErrorHandler` is built on top of the default error handler and offers the same functionality, but it’s tailored to support transacted routes. Chapter 12 focuses on transactions and discusses this error handler in detail, so we won’t say much about it here. For now, you just need to know that it exists and it’s a core part of Camel.

The remaining two error handlers are less commonly used and are much simpler.

11.2.4 THE NO ERROR HANDLER

`NoErrorHandler` is used to disable error handling. The current architecture of Camel mandates that an error handler must be configured, so if you want to disable error handling, you need to provide an error handler that's basically an empty shell with no real logic. That's `NoErrorHandler`.

11.2.5 THE LOGGING ERROR HANDLER

`LoggingErrorHandler` logs the failed message along with the exception. The logger uses standard log format from log kits such as log4j, commons logging, or the Java Util Logger.

Camel will, by default, log the failed message and the exception by using the log name `org.apache.camel.processor.LoggingErrorHandler` at the `ERROR` level. You can, of course, customize this.

The logging error handler can be replaced by using the dead letter channel error handler and using a log endpoint as the destination. Therefore, the logging error handler has been deprecated in favor of this approach.

That covers the five error handlers provided with Camel. Let's now look at the major features these error handlers provide.

11.2.6 FEATURES OF THE ERROR HANDLERS

The default, dead letter channel, and transaction error handlers are all built on the same base, `org.apache.camel.processor.RedeliveryErrorHandler`, so they all have several major features in common. These features are listed in table [11.2](#).

Table 11.2 Noteworthy features provided by the error handlers

Feature	Description

Redelivery policies	Redelivery policies allow you to indicate whether redelivery should be attempted. The policies also define settings such as the maximum number of redelivery attempts, delays between attempts, and so on.
Scope	Camel error handlers have two possible scopes: context (high level) and route (low level). The context scope allows you to reuse the same error handler for multiple routes, whereas the route scope is used for a single route only.
Exception policies	Exception policies allow you to define special policies for specific exceptions.
Error handling	This option allows you to specify whether the error handler should handle the error. You can let the error handler deal with the error or leave it for the caller to handle.

Let's continue by covering these noteworthy features. We'll start with redelivery and scope in section 11.3. Exception policies and error handling are covered in section 11.4.

11.3 Using error handlers with redelivery

Communicating with remote servers relies on network connectivity that can be unreliable and have outages. Luckily, these disruptions cause recoverable errors—the network connection could be reestablished in a matter of seconds or minutes. Remote services can also be the source of temporary problems, such as when the service is restarted by an administrator. To help address these problems, Camel supports a redelivery mechanism that allows you to control how recoverable errors are dealt with.

This section looks at a real-life error-handling scenario and then focuses on how Camel controls redelivery and how to configure and use it. We'll also look at using error handlers with

fault messages. We end this section by looking at error-handling scope and how it can be used to support multiple error handlers scoped at different levels.

11.3.1 AN ERROR-HANDLING USE CASE

Suppose you've developed an integration application at Rider Auto Parts that once every hour should upload files from a local directory to an HTTP server, and your boss asks why the files haven't been updated in the last few days. You're surprised, because the application has been running for the last month without a problem. This could well be a situation where neither error handling nor monitoring was in place.

Here's the Java file that contains the integration route:

```
from("file:/riders/files/upload?delay=15m")
    .to("http://riders.com/upload?
user=gear&password=secret");
```

This route will periodically scan for files in the /riders/files/upload folder, and if any files exist, it will upload them to the receiver's HTTP server using the HTTP endpoint.

But no explicit error handling is configured, so if an error occurs, the default error handler is triggered. That handler doesn't handle the exception but instead propagates it back to the caller. Because the caller is the file consumer, it'll log the exception and do a file rollback, meaning that any picked-up files will be left on the filesystem, ready to be picked up in the next scheduled poll.

At this point, you need to reconsider how errors should be handled in the application. You aren't in major trouble, because you haven't lost any files. Camel will move only successfully processed files out of the upload folder; failed files will just stack up.

The error occurs when sending the files to the HTTP server, so you look into the log files and quickly determine that Camel can't connect to the remote HTTP server because of network issues.

Your boss decides that the application should retry uploading the files if there's an error, so the files won't have to wait for the next hourly upload.

To implement this, you can configure the error handler to redeliver up to 5 times with 10-second delays:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(5)
    .redeliveryDelay(10000));
```

Configuring redelivery can hardly get any simpler than that. But let's take a closer look at how to use redelivery with Camel.

11.3.2 USING REDELIVERY

The first three error handlers in table 11.1 all support redelivery. This is implemented in the `RedeliveryErrorHandler` class, which they extend. `RedeliveryErrorHandler` must then know whether to attempt redelivery; that's what the redelivery policy is for.

A redelivery policy defines how and whether redelivery should be attempted. Table 11.3 outlines the most commonly used options supported by the redelivery policy and the default settings. The Camel team is working on a new website for the project that will overhaul the online documentation and include up-to-date documentation for all the redelivery options. Until then, you can find this information in the source code by looking at the `RoutePolicy` class:

<https://github.com/apache/camel/blob/master/camel-core/src/main/java/org/apache/camel/processor/RedeliveryPolicy.java>.

Table 11.3 Options provided in Camel for configuring redelivery

Option	Type	Default	Description
backoffMultiplier	d o u	. 0	Exponential back-off multiplier used to multiply each consequent delay. <code>redeliveryDelay</code> is the starting delay. Exponential back-off is disabled by default.

	b l e		
collis ionAvo idance Factor	d o u b l e	0 .1 5	A percentage to use when calculating a random delay offset (to avoid using the same delay at the next attempt). Will start with the <code>redeliveryDelay</code> as the starting delay. Collision avoidance is disabled by default.
logExh usted	b o o l e a n	t r u e	Specifies whether the exhaustion of redelivery attempts (when all redelivery attempts have failed) should be logged.
logExh usted Messag eBody	b o o l e a n	f a l s e	Specifies whether the message history should include the message body and headers. This is turned off by default to avoid logging content from the message payload, which can carry sensitive data.
logExh usted Messag eHisto ry	b o o l e a n		Specifies whether the message history should be logged when logging is exhausted. This option is by default <code>true</code> for all error handlers, except the dead letter channel, where it's <code>false</code> .
logRet ryAtte mpted	b o o l e a n	t r u e	Specifies whether redelivery attempts should be logged.
logSta ckTrace	b o o l e a n	t r u e	Specifies whether stacktraces should be logged when all redelivery attempts have failed.
maximu mRedel iveries	i n t	0	Maximum number of redelivery attempts allowed. 0 is used to disable redelivery, and -1 will attempt redelivery forever until it succeeds.

maximumRedeliveries	1 0 n o g 0 0	6	An upper bound in milliseconds for redelivery delay. This is used when you specify nonfixed delays, such as exponential back-off, to avoid the delay growing too large.
retryDelay	1 0 n o g 0 0	1	Fixed delay in milliseconds between each redelivery attempt.
useExponentialBackOff	b o o l e a n	f a l s e	Specifies whether exponential back-off is in use.

In the Java DSL, Camel has fluent builder methods for configuring the redelivery policy on the error handler. For instance, if you want to redeliver up to five times, use exponential back-off, and have Camel log at `WARN` level when it attempts a redelivery, you could use this code:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(5)
    .retryAttemptedLogLevel(LogLevel.WARN)
    .backOffMultiplier(2)
    .useExponentialBackOff());
```

Configuring this in XML DSL is done as follows:

```
<errorHandler id="myErrorHandler"
type="DefaultErrorHandler"
    <redeliveryPolicy maximumRedeliveries="5"
                    retryAttemptedLogLevel="WARN"
                    backOffMultiplier="2"
                    useExponentialBackOff="true"/>
</errorHandler>
```

Notice in XML DSL that you configure the redelivery policies by using the `<redeliveryPolicy>` element.

We've now established that Camel uses the information from the redelivery policy to determine whether and how to do redeliveries. But what happens inside Camel? As you'll recall

from figure 11.4, Camel includes a channel between every processing step in a route path, and there's functionality in these channels, such as error handlers. The error handler detects every exception that occurs and acts on it, deciding what to do, such as redeliver or give up.

Now that you know a lot about `DefaultErrorHandler`, it's time to try a little example.

11.3.3 USING DEFAULTERRORHANDLER WITH REDELIVERY

In the book's source code, you'll see an example in the `chapter11/errorhandler` directory. The example uses the following route configuration:

```
errorHandler(defaultErrorHandler())
```

①

①

Configures error handler

```
.maximumRedeliveries(2)  
.redeliveryDelay(1000)  
.retryAttemptedLogLevel(LoggingLevel.WARN));  
  
from("seda:queue.inbox")  
.bean("orderService", "validate")  
.bean("orderService", "enrich")
```

②

②

Invokes enrich method

```
.log("Received order ${body}")  
.to("mock:queue.order");
```

This configuration first defines a context-scoped error handler ① that will attempt at most two redeliveries using a one-second delay. When it attempts the redelivery, it will log this at the `WARN` level (as you'll see in a few seconds). The example is constructed to fail when the message reaches the `enrich` method ②.

And here's the example using XML DSL instead:

```
<camelContext errorHandlerRef="myErrorHandler">
    <errorHandler id="myErrorHandler"
type="DefaultErrorHandler">
        <redeliveryPolicy maximumRedeliveries="2"
                           redeliveryDelay="1000"
                           retryAttemptedLogLevel="WARN"/>
    </errorHandler>

    <route>
        <from uri="seda:queue.inbox"/>
        <bean ref="orderService" method="validate"/>
        <bean ref="orderService" method="enrich"/>
        <log message="Received order ${body}"/>
        <to uri="mock:queue.order"/>
    </route>
</camelContext>
```

You can run this example by using the following Maven goal from the chapter11/errorhandler directory:

```
mvn test -Dtest=DefaultErrorHandlerTest
mvn test -Dtest=SpringDefaultErrorHandlerTest
```

When running the example, you'll see the following log entries outputted on the console. Notice how Camel logs the redelivery attempts:

```
2017-03-08 14:28:16,959 [a://queue.inbox] WARN
DefaultErrorHandler - Failed delivery for (MessageId: ID-
davsclaus-pro-local-1512...). On delivery attempt: 0
caught: camelinaction.OrderException: ActiveMQ in Action is
out of stock
2017-03-08 14:28:17,960 [a://queue.inbox] WARN
DefaultErrorHandler - Failed delivery for (MessageId: ID-
davsclaus-pro-local-1512...). On delivery attempt: 1
caught: camelinaction.OrderException: ActiveMQ in Action is
out of stock
```

These log entries show that Camel failed to deliver a message, which means the entry is logged after the attempt is made. On delivery attempt: 0 identifies the first attempt; attempt 1 is the first redelivery attempt. Camel also logs the exchangeId (which you can use to correlate messages) and the exception that caused

the problem (without the stacktrace, by default).

When Camel performs a redelivery attempt, it does so at the point of origin. In the preceding example, the error occurs when invoking the `enrich` method ②, which means Camel will redeliver by retrying the `.bean("orderService", "enrich")` step in the route.

Camel redelivery happens at the point of error

It's important to understand that Camel's error handler performs redelivery at the point of error, and not from the beginning of the route. When an exception happens during routing, Camel doesn't *roll back* the entire exchange and start all over again. No, Camel performs the redelivery at the *same point* where the error happened. The next chapter covers transactions, including the concept of rolling back a transaction and restarting the transaction process all over again. But you shouldn't mix up these concepts with transactions using Camel's error handler. They aren't the same. That said, tricks can allow you to let Camel's error handler redeliver an entire route in case an exception was thrown anywhere from the route. Section 11.4.7 covers such an example.

After all redelivery attempts have failed, we say it's *exhausted*, and Camel logs this at the `ERROR` level by default. (You can customize this with the options listed in table 11.3.) When the redelivery attempts are exhausted, the log entry is similar to the previous ones, but Camel explains that it's exhausted after three attempts:

```
2017-03-08 14:28:18,961 [a://queue.inbox] ERROR  
DefaultErrorHandler -  
Failed delivery for (MessageId: ID-davsclaus-pro-local-
```

```
1512...).  
Exhausted after delivery attempt: 3 caught:  
camelinaction.OrderException:  
ActiveMQ in Action is out of stock
```

In addition to logging the exception message, Camel also logs a *route trace* that prints each step from all the routes the message had taken up until this error, as shown here:

Message History

```
-----  
RouteId ProcessorId Processor  
Elapsed (ms)  
[route2] [route2] [seda://queue.inbox ]  
[ 3009]  
[route2] [bean3] [bean[ref:orderService method: val]  
[ 1]  
[route2] [bean4] [bean[ref:orderService method: enh]  
[ 2007]
```

Here you can see that the exception occurred at the last step, which is at ID bean4, which would be calling the `orderService` bean and the `enrich` method. Further below is the full stacktrace of the caused exception:

Stacktrace

```
-----  
camelinaction.OrderException: ActiveMQ in Action is out of  
stock  
at  
camelinaction.OrderService.enrich(OrderService.java:22)  
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native  
Method)  
at  
org.apache.camel.component.bean.MethodInfo.invoke(MethodInf  
o.java:478)
```

As you can see, detailed information from Camel is provided when an exception occurs, and the exception can't be mitigated by the error handler (all redelivery attempts have failed, and the exchange is regarded as exhausted).

The level of information can be turned down so that it's only

the exception being logged by setting the option log exhausted message history to `false`:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(2)
    .redeliveryDelay(1000)
    .retryAttemptedLogLevel(LoggingLevel.WARN)
    .logExhaustedMessageHistory(false);
```

And here's how to configure it using XML DSL:

```
<errorHandler id="myErrorHandler"
type="DefaultErrorHandler">
    <redeliveryPolicy redeliveryDelay="1000"
                      retryAttemptedLogLevel="WARN"
                      logExhaustedMessageHistory="false"/>
</errorHandler>
```

TIP The default error handler has many options, which are listed in table 11.3. We encourage you to try loading this example into your IDE and playing with it. Change the settings on the error handler and see what happens.

Unlike the default error handler, the dead letter channel won't log any activity by default. But you can configure the dead letter channel to do so, such as setting `logExhaustedMessageHistory=true` to enable the detailed logging.

The preceding log output identifies the number of redelivery attempts, but how does Camel know this? Camel stores this information on the exchange. Table 11.4 reveals where this information is stored.

Table 11.4 Headers on the exchange related to error handling

Header	Type	Description
<code>Exchange.REDELIVERY_COUNTER</code>	int	The current redelivery attempt
<code>Exchange.REDELIVERED</code>	bool	Whether this exchange is being redelivered

RED	leain	
Exchange.REDELIVERY_EXHAUSTED	bool leain	Whether this exchange has attempted (exhausted) all redeliveries and has still failed

The information in table 11.4 is available only when Camel performs a redelivery; these headers are absent on the regular first attempt. It's only when a redelivery is triggered that these headers are set on the exchange.

11.3.4 ERROR HANDLERS AND SCOPES

Scopes can be used to define error handlers at different levels. Camel supports two scopes: a context scope and a route scope.

IMPORTANT When using Java DSL, context scope is implemented as per the RouteBuilder scope, but can become a global scope by letting all RouteBuilder classes inherit from a base class where the context-scoped error handler is configured. You'll see this in action later in this section.

Camel allows you to define a global context-scoped error handler that's used by default, and, if needed, you can also configure a route-scoped error handler that applies only to a particular route. This is illustrated in the following listing.

Listing 11.3 Using two error handlers at different scopes

```
errorHandler(defaultErrorHandler()) ①
```

①

Defines context-scoped error handler

```
.maximumRedeliveries(2)
.redeliveryDelay(1000)
.retryAttemptedLogLevel(LogLevel.WARN));
```

```
from("file://target/orders?delay=10000")
    .bean("orderService", "toCsv")
    .to("mock:file")
    .to("seda:queue.inbox");

from("seda:queue.inbox")
    .errorHandler(deadLetterChannel("log:DLC")) ②
```

②

Defines route-scoped error handler

```
.maximumRedeliveries(5).retryAttemptedLogLevel(LogLevel
.INFO)
    .redeliveryDelay(250).backOffMultiplier(2))
    .bean("orderService", "validate")
    .bean("orderService", "enrich") ③
```

③

Invokes enrich method

```
.to("mock:queue.order");
```

Listing 11.3 is an improvement over the previous error-handling example. The default error handler is configured as in the previous example **①**, but you have a new route that picks up files, processes them, and sends them to the second route. This first route will use the default error handler **①** because it doesn't have a route-scoped error handler configured, but the second route has a route-scoped error handler **②**. It's a DeadLetterChannel that will send failed messages to a log. Notice that it has different options configured than the former error handler.

The book's source code includes this example, which you can run by using the following Maven goal from the chapter11/errorhandler directory:

```
mvn test -Dtest=RouteScopeTest
```

This example should fail for some messages when the enrich

method ❸ is invoked. This demonstrates how the route-scoped error handler is used as an error handler.

The most interesting part of this test class is the `testOrderActiveMQ` method (see the example with the source code), which will fail in the second route and therefore show the dead letter channel in action. There are a couple of things to notice about this, such as the exponential back-off, which causes Camel to double the delay between redelivery attempts, starting with 250 milliseconds and ending with 4 seconds.

The following snippets show what happens at the end when the error handler is exhausted:

```
2017-03-08 17:03:44,534 [a://queue.inbox] INFO DeadLetterChannel - Failed delivery for (MessageId: ID-davsclaus-pro-local-1512...). On delivery attempt: 5 caught: camelinaction.OrderException: ActiveMQ in Action is out of stock
2017-03-08 17:03:44,541 [a://queue.inbox] INFO DLC - Exchange[BodyType:String, Body:amount=1, name=ActiveMQ in Action, id=123]
2017-03-08 17:03:44,542 [a://queue.inbox] ERROR DeadLetterChannel - Failed delivery for (MessageId: ID-davsclaus-pro-local-1512...). Exhausted after delivery attempt: 6 caught: camelinaction.OrderException: ActiveMQ in Action is out of stock. Processed by failure processor: sendTo(Endpoint[log://DLC])
```

As you can see, the dead letter channel moves the message to its dead letter queue, which is the `log://DLC` endpoint. After this, Camel also logs an `ERROR` line indicating that this move was performed.

We encourage you to try this example and adjust the configuration settings on the error handlers to see what happens.

So far, the error-handling examples in this section have used the Java DSL. Let's look at configuring error handling with XML DSL.

USING ERROR HANDLING WITH XML DSL

Let's revise the example in [listing 11.3](#) to use XML DSL. Here's how that's done.

[Listing 11.4](#) Using error handling with XML DSL

```
<bean id="orderService"
      class="camelaction.OrderService"/>

<camelContext id="camel" errorHandlerRef="defaultEH" ❶

❶ Specifies context-scoped error handler

xmlns="http://camel.apache.org/schema/spring">

<errorHandler id="defaultEH"> ❷

❷ Sets up context-scoped error handler

    <redeliveryPolicy maximumRedeliveries="2"
    redeliveryDelay="1000"
                      retryAttemptedLogLevel="WARN"/>
  </errorHandler>

<errorHandler id="dlc" ❸

❸ Sets up route-scoped error handler

    type="DeadLetterChannel"
  deadLetterUri="log:DLC">
    <redeliveryPolicy maximumRedeliveries="5"
    redeliveryDelay="250"
                      retryAttemptedLogLevel="INFO"
                      backOffMultiplier="2"
  useExponentialBackOff="true"/>
  </errorHandler>
```

```
<route>
  <from uri="file://target/orders?delay=10000"/>
  <bean ref="orderService" method="toCsv"/>
  <to uri="mock:file"/>
  <to uri="seda:queue.inbox"/>
</route>

<route errorHandlerRef="dlc"> ④
```

④

Specifies route-scoped error handler

```
<from uri="seda:queue.inbox"/>
<bean ref="orderService" method="validate"/>
<bean ref="orderService" method="enrich"/>
<to uri="mock:queue.order"/>
</route>
</camelContext>
```

To use a context-scoped error handler in XML DSL, you must configure it by using an `errorHandlerRef` attribute ① on the `camelContext` tag. The `errorHandlerRef` refers to an `<errorHandler>`, which in this case is the default error handler with ID "defaultEH" ②. There's another error handler, a `DeadLetterChannel` error handler ③, that's used at route scope in the second route ④.

As you can see, the differences between the Java DSL and XML DSL mostly result from using the `errorHandlerRef` attribute to reference the error handlers in XML DSL, whereas Java DSL can have route-scoped error handlers within the routes.

You can try this example by running the following Maven goal from the chapter11/errorhandler directory:

```
mvn test -Dtest=SpringRouteScopeTest
```

The XML DSL file is located in the `src/test/resources/camelinaction` directory.

Now we've covered how to use error handlers in Java and XML DSL, but there's a subtle difference when using error handlers in the two DSLs. In XML DSL, context-scoped error

handlers are reusable among all the routes as is. But in Java DSL, you must use an abstract base class, which your `RouteBuilder` classes extend to reuse context-scoped error handlers. The following section highlights the difference.

11.3.5 REUSING CONTEXT-SCOPED ERROR HANDLERS

Reusing a context-scoped error handler in XML DSL is straightforward, as you'd expect. The following listing shows how easy that is.

Listing 11.5 Reusing context-scoped error handler in XML DSL

```
<bean id="orderService"
      class="camelaction.OrderService"/>
<camelContext id="camel" errorHandlerRef="defaultEH" ①
```

1

Specifies context-scoped error handler

```
xmlns="http://camel.apache.org/schema/spring">
```

```
  <errorHandler id="defaultEH"> ②
```

2

Sets up context-scoped error handler

```
    <redeliveryPolicy maximumRedeliveries="2"
      redeliveryDelay="1000"          retryAttemptedLogLevel="WARN"/>
  </errorHandler>
```

```
  <route> ③
```

3

Route using context-scoped error handler out of the box

```
<from uri="file://target/orders?delay=10000"/>
<bean ref="orderService" method="toCsv"/>
<to uri="mock:file"/>
<to uri="seda:queue.inbox"/>
</route>

<route> 4
```

4

Another route using context-scoped error handler out of the box

```
<from uri="seda:queue.inbox"/>
<bean ref="orderService" method="validate"/>
<bean ref="orderService" method="enrich"/>
<to uri="mock:queue.order"/>
</route>
</camelContext>
```

In XML DSL, you configure the context-scoped error handler on `<camelContext>` by using the `errorHandlerRef` attribute **1**. The error handler is then configured by using the `<errorHandler>` tag **2**. And all routes (**3** **4**) will automatically use the context-scoped error handler by default.

In Java DSL, the situation is a bit different, as the context-scoped error handler is essentially scoped to its `RouteBuilder` instance. The same example from [listing 11.5](#) has been ported to Java DSL in the following listing.

[Listing 11.6](#) Reusing context-scoped error handler in Java DSL

```
public abstract class BaseRouteBuilder extends RouteBuilder
{ 1
```

1

Base class to hold the reused error handler

```
    public void configure() throws Exception {
        errorHandler(deadLetterChannel("mock:dead")) 2
```

2

2

The context-scoped error handler to reuse

```
.maximumRedeliveries(2)
.redeliveryDelay(1000)
.retryAttemptedLogLevel(LoggingLevel.WARN));
}
}

public class InboxRouteBuilder extends BaseRouteBuilder
{ ③
```

3

RouteBuilder extending the base class

```
public void configure() throws Exception {
    super.configure(); ④
```

4

Calling super.configure() to reuse the context-scoped error handler

```
from("file://target/orders?delay=10000") ⑤
```

5

Route that will use the context-scoped error handler

```
.bean("orderService", "toCsv")
.to("mock:file")
.to("seda:queue.inbox");
}
}

public class OrderRouteBuilder extends BaseRouteBuilder
{ ⑥
```

6

Another RouteBuilder extending the base class

```
public void configure() throws Exception {  
    super.configure();
```

7

Calling super.configure() to reuse the context-scoped error handler

```
from("seda:queue.inbox")
```

8

Route that also will use the context-scoped error handler

```
.bean("orderService", "validate")  
.bean("orderService", "enrich")  
.to("mock:queue.order");  
}  
}
```

In Java DSL, you'd need to use object-oriented principles such as inheritance to reuse the context-scoped error handler. Therefore, you create a base class ①, where you configure the context-scoped error handler ② in the `configure` method. Our `RouteBuilder` classes now must extend the base class (③ ⑥) and call the `super.configure` method (④ ⑦), which calls the parent class that has the error handler to reuse. Each of the routes (⑤ ⑧) will now use the context-scoped error handler that you defined in the base class ②.

As you can see, the subtle difference is that in Java DSL you must rely on inheritance and must remember to call the `super.configure` method.

We encourage you to try this example in action with the source code from the book. The example is in the `chapter11/reuse` directory, and you can try the example with the following Maven goal:

```
mvn test -Dtest=ReuseErrorHandlerTest  
mvn test -Dtest=SpringReuseErrorHandlerTest
```

Try to experiment and see what happens if you forget to call the `super.configure` method in the `OrderRouteBuilder` class.

This concludes our discussion of scopes and redelivery.

We'll continue in the next section to look at the other two major features that error handlers provide, as listed in table 11.2: exception policies and error handling. This is a good time to take a break before continuing. Go grab a fresh cup of coffee or tea; or maybe it's time to walk the dog or empty the dishwasher.

11.4 Using exception policies

Exception policies are used to intercept and handle specific exceptions in particular ways. For example, exception policies can influence, at runtime, the redelivery policies the error handler is using. They can also handle an exception or even detour a message.

NOTE In Camel, exception policies are specified with the `onException` method in the route, so we use the term `onException` interchangeably with *exception policy*.

We'll cover exception policies piece by piece, looking at how they catch exceptions, how they work with redelivery, and how they handle exceptions. Then we'll take a look at custom error handling and put it all to work in an example.

11.4.1 UNDERSTANDING HOW ONEXCEPTION CATCHES EXCEPTIONS

We'll start by looking at how Camel inspects the exception hierarchy to determine how to handle the error. This will give you a better understanding of how you can use `onException` to your advantage.

Imagine you have this exception with underlying wrapped exceptions being thrown:

```
org.apache.camel.RuntimeCamelException (wrapper by Camel)
+ com.mycompany.OrderFailedException
  + java.net.ConnectException
```

The real cause is a `ConnectException`, but it's wrapped in an `OrderFailedException` and yet again in a `RuntimeCamelException`.

Camel will traverse the hierarchy from the bottom up to the root searching for an `onException` that matches the exception. In this case, Camel will start with `java.net.ConnectException`, move on to `com.mycompany.OrderFailedException`, and finally reach `RuntimeCamelException`. For each of those three exceptions, Camel will compare the exception to the defined `onExceptions` to select the best matching `onException` policy. If no suitable policy can be found, Camel relies on the configured error-handler settings. You'll drill down and look at how the matching works, but for now you can think of this as Camel doing a big `instanceof` check against the exceptions in the hierarchies, following the order in which the `onExceptions` were defined.

Suppose you have a route with the following `onException`:

```
onException(OrderFailedException.class).maximumRedeliveries(3);
```

The aforementioned `ConnectException` is being thrown, and the Camel error handler is trying to handle this exception. Because you have an exception policy defined, it will check whether the policy matches the thrown exception. The matching is done as follows:

1. Camel starts with the `java.net.ConnectException` and compares it to `onException(OrderFailedException.class)`. Camel checks whether the two exceptions are exactly the same type, and in this case they're not—`ConnectionException` and `OrderFailedException` aren't the same type.
2. Camel checks whether `ConnectException` is a subclass of `OrderFailedException`, and this isn't true either. So far, Camel hasn't found a match.
3. Camel moves up the exception tree (wrapped by) and compares

again with `OrderFailedException`. This time there's an exact match, because they're both of the type `OrderFailedException`.

No more matching takes place. Camel got an exact match, and the exception policy will be used.

When an exception policy has been selected, its configured policy will be used by the error handler. In this example, the policy defines the maximum redeliveries to be three, so the error handler will attempt at most three redeliveries when this kind of exception is thrown.

Any value configured on the exception policy will override options configured on the error handler. For example, suppose the error handler had the `maximumRedeliveries` option configured as 5. Because the `onException` has the same option configured, its value of 3 will be used instead.

IMPORTANT The book's source code has an example that demonstrates what you've just learned. Take a look at the `OnExceptionTest` class in `chapter11/onexception`. It has multiple test methods (`testOnExceptionDirectMatch` and `testOnExceptionWrappedMatch`), each showing a scenario of how `onException` works. You can run the tests by using Maven from the command shell: `mvn test -Dtest=OnExceptionTest`.

Let's make the example a bit more interesting and add a second `onException` definition:

```
onException(OrderFailedException.class).maximumRedeliveries  
(3);  
onException(ConnectException.class).maximumRedeliveries(10)  
;
```

If the same exception hierarchy is thrown as in the previous example, Camel would select the second `onException` because it directly matches `ConnectionException`. This allows you to define different strategies for different kinds of exceptions. In this example, it's configured to use more redelivery attempts for

connection exceptions than for order failures.

TIP This example demonstrates how `onException` can influence the redelivery policies the error handler uses. If an error handler were configured to perform only three redelivery attempts, the preceding `onException` would overload this with 10 redelivery attempts in the case of connection exceptions.

But what if there are no direct matches? Let's look at another example. This time, imagine that a `java.io.IOException` exception is thrown. Camel will do its matching, and because `OrderFailedException` isn't a direct match, and `IOException` isn't a subclass of it, it's out of the game. The same applies for `ConnectException`. In this case, there are no `onException` definitions that match, and Camel will fall back to using the configuration of the current error handler.

You can see this in action by running the following Maven goal from the chapter 11/onexception directory:

```
mvn test -Dtest=OnExceptionFallbackTest
```

ONEXCEPTION AND GAP DETECTION

Can Camel do better if there isn't a direct hit? Yes, because Camel uses a gap-detection mechanism that calculates the gaps between a thrown exception and the `onExceptions` and then selects the `onException` with the lowest gap as the winner. An example will explain this better.

Suppose you have these three `onException` definitions, each having a different redelivery policy:

```
onException(ConnectException.class)
    .maximumRedeliveries(5);

onException(IOException.class)
    .maximumRedeliveries(3).redeliveryDelay(1000);
```

```
onException(Exception.class)
    .maximumRedeliveries(1).redeliveryDelay(5000);
```

And imagine this exception is thrown:

```
org.apache.camel.OrderFailedException
+ java.io.FileNotFoundException
```

Which of those three `onExceptions` would be selected?

Camel starts with `java.io.FileNotFoundException` and compares it to the `onException` definitions. Because there are no direct matches, Camel uses gap detection. In this example, only `onException(IOException.class)` and `onException(Exception.class)` partly match, because `java.io.FileNotFoundException` is a subclass of `java.io.IOException` and `java.lang.Exception`.

Here's the exception inheritance hierarchy for `FileNotFoundException`:

```
java.lang.Exception
+ java.io.IOException
  + java.io.FileNotFoundException
```

Looking at this exception hierarchy, you can see that `java.io.FileNotFoundException` is a direct subclass of `java.io.Exception`, so the gap is computed as 1. The gap between `java.lang.Exception` and `java.io.FileNotFoundException` is 2. At this point, the best candidate has a gap of 1.

Camel will then follow the same process with the next exception from the thrown exception hierarchy, which is `OrderFailedException`. This time, it's only the `onException(Exception.class)` that partly matches, and the gap between `OrderFailedException` and `Exception` is also 1:

```
java.lang.Exception
+ OrderNotFoundException
```

What now? You have two gaps, both calculated as 1. In the case of a tie, Camel will always pick the first match, because the cause exception is most likely the last in the hierarchy. In this case, it's

a `FileNotFoundException`, so the winner will be `onException(IOException.class)`.

This example is provided in the source code for the book in the `chapter11/onexception` directory. You can try it using the following Maven goal:

```
mvn test -Dtest=OnExceptionGapTest  
mvn test -Dtest=SpringOnExceptionGapTest
```

MULTIPLE EXCEPTIONS PER ONEXCEPTION

So far, you've seen only examples with one exception per `onException`, but you can define multiple exceptions in the same `onException`:

```
onException(XPathException.class,  
TransformerException.class)  
    .to("log:xml?level=WARN");  
onException(IOException.class, SQLException.class,  
JMSEException.class)  
    .maximumRedeliveries(5).redeliveryDelay(3000);
```

Here's the same example using XML DSL:

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
    <onException>  
  
        <exception>javax.xml.xpath.XPathException</exception>  
  
        <exception>javax.xml.transform.TransformerException</exception>  
            <to uri="log:xml?level=WARN"/>  
        </onException>  
        <onException>  
            <exception>java.io.IOException</exception>  
            <exception>java.sql.SQLException</exception>  
            <exception>javax.jms.JMSEException</exception>  
            <redeliveryPolicy maximumRedeliveries="5"  
redeliveryDelay="3000"/>  
        </onException>  
    </camelContext>
```

Our next topic is how `onException` works with redelivery. Even

though we've touched on this already in our examples, we'll go into the details in the next section.

11.4.2 UNDERSTANDING HOW ONEXCEPTION WORKS WITH REDELIVERY

`onException` works with redeliveries, but you need to be aware of a couple of things that might not be immediately obvious.

Suppose you have the following route:

```
from("jetty:http://0.0.0.0/orderservice")
    .to("netty4:tcp://erp.rider.com:4444?textline=true")
    .bean("orderBean", "prepareReply");
```

You use the Camel Jetty component to expose an HTTP service where statuses of pending orders can be queried. The order status information is retrieved from a remote ERP system by the Netty component using low-level socket communication. You've learned how to configure this on the error handler itself, but it's also possible to configure this on `onException`.

Suppose you want Camel to retry invoking the external TCP service, in case there has been an I/O-related error, such as a lost network connection. To do this, you can add `onException` and configure the redelivery policy as you like. In the following example, the redelivery tries at most five times:

```
onException(IOException.class).maximumRedeliveries(5);
```

You've already learned that `onException(IOException.class)` will catch those I/O-related exceptions and act accordingly. But what about the delay between redeliveries?

In this example, the delay will be 1 second. Camel will use the default redelivery policy settings outlined in table 11.3 and then override those values with values defined in `onException`. Because the delay wasn't overridden in `onException`, the default value of 1 second is used.

TIP When you configure redelivery policies, they override the

existing redelivery policies set in the current error handler. This is convention over configuration, because you need to configure only the differences, which is often just the number of redelivery attempts or a different redelivery delay.

Now let's make it a bit more complicated:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .redeliveryDelay(2000));

onException(IOException.class).maximumRedeliveries(5);

from("jetty:http://0.0.0.0/orderservice")
    .to("netty4:tcp://erp.rider.com:4444?textline=true")
    .bean("orderBean", "prepareReply");
```

What would the redelivery delay be if `IOException` were thrown? Yes, it's 2 seconds, because `onException` will fall back and use the redelivery policies defined by the error handler, and its value is configured as `redeliveryDelay(2000)`.

Now let's remove the `maximumRedeliveries(5)` option from `onException`, so it's defined as `onException(IOException.class)`:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .redeliveryDelay(2000));

onException(IOException.class);

from("jetty:http://0.0.0.0/orderservice")
    .to("netty4:tcp://erp.rider.com:4444?textline=true")
    .bean("orderBean", "prepareReply");
```

What would the maximum number of redeliveries be now, if `IOException` were thrown? You'll probably say the answer is 3—the value defined on the error handler. In this case, though, the answer is 0. Camel won't attempt any redelivery because any `onException` will override the `maximumRedeliveries` to 0 by default (redelivery is disabled by default) unless you explicitly set the `maximumRedeliveries` option.

The reason Camel implements this behavior is our next topic: using `onException` to handle exceptions.

11.4.3 UNDERSTANDING HOW ONEXCEPTION CAN HANDLE EXCEPTIONS

Suppose you have a complex route that processes a message in multiple steps. Each step does some work on the message, but any step can throw an exception to indicate that the message can't be processed and that it should be discarded. This is where handling exceptions with `onException` comes into the game.

Handling an exception with `onException` is similar to exception handling in Java itself. You can think of it as being like using a `try ... catch` block.

This is best illustrated with an example. Imagine you need to implement an ERP server-side service that serves order statuses. This is the ERP service you called from the previous section:

```
public void configure() {  
    try {  
        from("netty4:tcp://0.0.0.0:4444?textline=true")  
            .process(new ValidateOrderId())  
            .to("jms:queue:order.status")  
            .process(new GenerateResponse());  
    } catch (JMSEException e) {  
        process(new GenerateFailureResponse()); 1  
    }  
}
```

1

Rethrows caught exception

```
}  
}
```

This snippet of pseudocode involves multiple steps in generating the response. If something goes wrong, you catch the exception and return a failure response 1.

We call this *pseudocode* because it shows your intention but the code won't compile. This is because the Java DSL uses the fluent builder syntax, where method calls are stacked together to

define the route. The regular `try ... catch` mechanism in Java works at runtime to catch exceptions that are thrown when the `configure` method is executed, but in this case the `configure` method is invoked only once, when Camel is started (when it initializes and builds up the route path to use at runtime).

Don't despair. Camel has a counterpart to the classic `try ... catch ... finally` block in its DSL: `doTry ... doCatch ... doFinally`.

USING `doTry`, `doCatch`, AND `doFinally`

The following listing shows how to make the code compile and work at runtime as you would expect with a `try ... catch` block.

[Listing 11.7](#) Using `doTry ... doCatch` with Camel routing

```
public void configure() {
    from("netty4:tcp://0.0.0.0:4444?textline=true")
        .doTry()
            .process(new ValidateOrderId())
            .to("jms:queue:order.status")
            .process(new GenerateResponse())
        .doCatch(JMSEException.class)
            .process(new GenerateFailureResponse())
        .end();
}
```

The `doTry ... doCatch` block is a bit of a sidetrack, but it's useful because it helps bridge the gap between thinking in regular Java code and thinking in EIPs.

USING `onException` TO HANDLE EXCEPTIONS

The `doTry ... doCatch` block has one limitation: it's only route-scoped. The blocks work only in the route in which they're defined. `onException`, on the other hand, works in both context and route scopes, so let's try revising the last listing using `onException`. That's illustrated in the following listing.

[Listing 11.8](#) Using `onException` in context scope

```
onException(JMSEException.class)
    .handled(true) ①
```

①

Handles all JMSEExceptions

```
.process(new GenerateFailureResponse()));

from("netty4:tcp://0.0.0.0:4444?textline=true")
    .process(new ValidateOrderId())
    .to("jms:queue:order.status")
    .process(new GenerateResponse());
```

A difference between `doCatch` and `onException` is that `doCatch` will handle the exception, whereas `onException`, by default, won't handle it. That's why you use `handled(true)` ① to instruct Camel to handle this exception. As a result, when `JMSEException` is thrown, the application acts as if the exception were caught in a catch block using the regular Java `try ... catch` mechanism.

In [listing 11.8](#), you should also notice that the concerns are separated and the normal route path is laid out nicely and simply; it isn't mixed up with the exception handling.

Imagine that a message arrives on the TCP endpoint, and the Camel application routes the message. The message passes the validate processor and is about to be sent to the JMS queue, but this operation fails and `JMSEException` is thrown. [Figure 11.5](#) is a sequence diagram showing the steps that take place inside Camel in such a situation. It shows how `onException` is triggered to handle the exception.

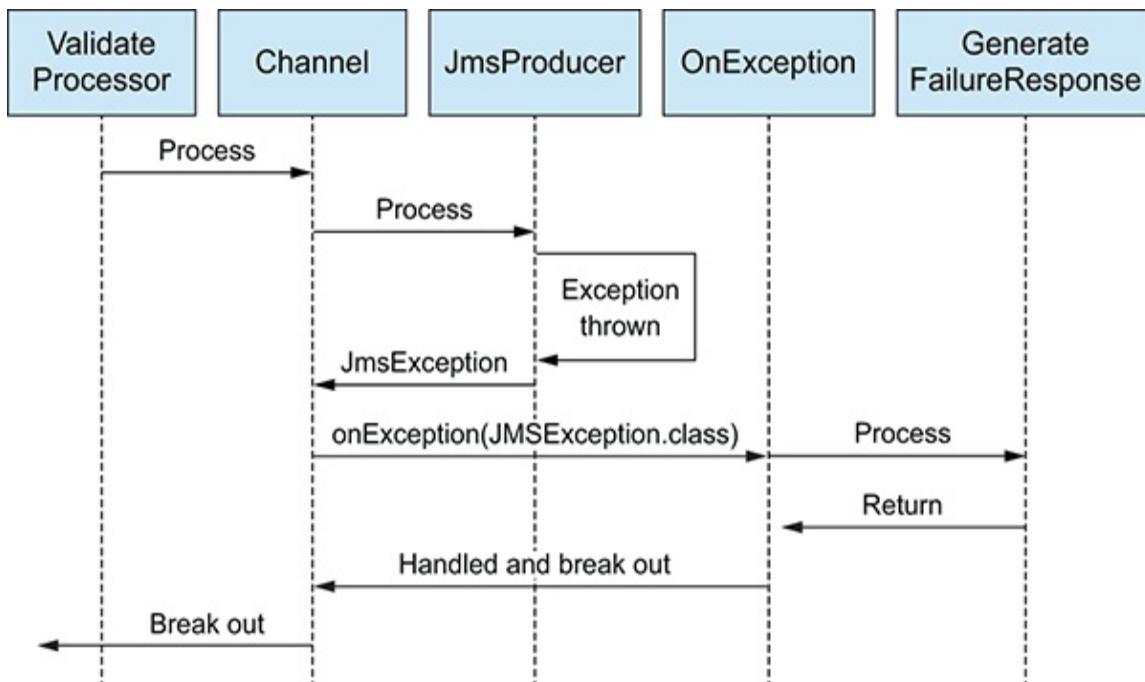


Figure 11.5 Sequence diagram of a message being routed and JMSEexception being thrown from JmsProducer, which is handled by onException. onException generates a failure that's returned to the caller.

Figure 11.5 shows how JmsProducer throws JMSEexception to Channel1, which is where the error handler lives. The route has onException defined that reacts when JMSEexception is thrown, and it processes the message. The GenerateFailureResponse processor generates a custom failure message that's supposed to be returned to the caller. Because onException was configured to handle exceptions handled(true), Camel will *break out* from continuing the routing and will return the failure message to the initial consumer, which in turn returns the custom reply message.

NOTE onException doesn't handle exceptions by default, so listing 11.8 uses handled(true) to indicate that onException should handle the exception. This is important to remember, because it must be specified when you want to handle the exception. Handling an exception won't continue routing from the point where the exception was thrown. Camel will break out of the route and continue routing on onException. If you want to

ignore the exception and continue routing, you must use `continued(true)`, which is discussed in section 11.4.6.

Before we move on, let's take a minute to look at the example from listing 11.8 revised to use XML DSL. The syntax is a bit different, as you can see.

Listing 11.9 XML DSL revision of listing 11.8

```
<camelContext
    xmlns="http://camel.apache.org/schemas/spring">
    <onException>
        <exception>javax.jms.JMSEException</exception>
        <handled><constant>true</constant></handled> ①
```

①

Handles all JMSEExceptions

```
        <process ref="failureResponse"/>
    </onException>

    <route>
        <from uri="netty4:tcp://0.0.0.0:4444?
textline=true"/>
        <process ref="validateOrder"/>
        <to uri="jms:queue:order.status"/>
        <process ref="generateResponse"/>
    </route>
</camelContext>
```

```
<bean id="failureResponse" ②
```

②

Processor generates failure response

```
        class="camelaction.FailureResponseProcessor"/>
<bean id="validateOrder"
    class="camelaction.ValidateProcessor"/>
<bean id="generateResponse"
    class="camelaction.ResponseProcessor"/>
```

Notice how `onException` is set up—you must define the exceptions in the `exception` tag. Also, `handled(true)` ❶ is a bit longer because you must enclose it in the `<constant>` expression. There are no other noteworthy differences in the rest of the route.

Listing 11.9 uses a custom processor to generate a failure response ❷. Let's take a closer look at that.

11.4.4 CUSTOM EXCEPTION HANDLING

Suppose you want to return a custom failure message, as in listing 11.9, that not only indicates what the problem was, but also includes details from the current exchange. How can you do that?

Listing 11.9 laid out how to do this by using `onException`. The following listing shows how the failure Processor could be implemented.

Listing 11.10 Using a processor to create a failure response to be returned to the caller

```
public class FailureResponseProcessor implements Processor
{
    public void process(Exchange exchange) throws Exception
    {
        String body =
exchange.getIn().getBody(String.class);
        Exception e =
exchange.getProperty(Exchange.EXCEPTION_CAUGHT,
                    Exception.class); ❶
```

❶

Gets the exception

```
StringBuilder sb = new StringBuilder();
sb.append("ERROR: ");
sb.append(e.getMessage());
sb.append("\nBODY: ");
sb.append(body);
exchange.getIn().setBody(sb.toString());
```

```
    }  
}
```

First, you grab the information you need: the message body and the exception ❶. It may seem a bit odd that you get the exception as a property instead of `exchange.getException()`. You do that because you've marked `onException` to handle the exception; this was done at ❶ in [listing 11.9](#). Consequently, Camel moves the exception from the Exchange to the `Exchange.EXCEPTION_CAUGHT` property. The rest of the processor builds the custom failure message that's to be returned to the caller.

You may wonder whether there are other properties Camel sets during error handling, and there are. They're listed in table [11.5](#). But from an end-user perspective, only the first two properties in table [11.5](#) matter. The other two properties are used internally by Camel in its error handling and routing engine.

One example of when the `FAILURE_ENDPOINT` property comes in handy is when you route messages through the Recipient List EIP, which sends a copy of the message to a dynamic number of endpoints. Without this information, you wouldn't know precisely which of those endpoints failed.

Table 11.5 Properties on the exchange related to error handling

Property	Type	Description
<code>Exchange.EXCEPTION_CAUGHT</code>	<code>Exception</code>	The exception that was caught.
<code>Exchange.FAILURE_ENDPOINT</code>	<code>String</code>	The URL of the endpoint that failed if a failure occurred when sending to an endpoint. If the failure didn't occur while sending to an endpoint, this property is <code>null</code> .

Exchange.E RRORHANDLE R_HANDLED	b o o l e a n	Whether the error handler handled the exception.
Exchange.F AILURE_HAN DLED	b o o l e a n	Whether <code>onException</code> handled the exception. Or true if the exchange was moved to a dead letter queue.

It's worth noting that in [listing 11.10](#) you use a Camel Processor, which forces you to depend on the Camel API. You can use a bean instead, as shown in the following listing.

Listing 11.11 Using a bean to create a failure response to be returned to the caller

```
public class FailureResponseBean {
    public String failMessage(String body, Exception e)
{ ①
```

①

Exception provided as parameter

```
        StringBuilder sb = new StringBuilder();
        sb.append("ERROR: ");
        sb.append(e.getMessage());
        sb.append("\nBODY: ");
        sb.append(body);
        return sb.toString();
    }
}
```

As you can see, you can use Camel's parameter binding ① to declare the parameter types you want to use. The first parameter is the message body, and the second is the exception.

Suppose you use a custom processor or bean to create a failure response. What would happen if a new exception were thrown

from the processor or bean? That's the topic for the next section.

11.4.5 NEW EXCEPTION WHILE HANDLING EXCEPTION

This section looks at how Camel reacts when a new exception occurs while handling a previous exception, which has been constructed as an example in the following listing.

Listing 11.12 Using onException in context scope

```
onException(AuthorizationException.class) ①
```

①

onException to handle AuthorizationException

```
.handled(true)  
.process(new NotAllowedProcessor()); ②
```

②

By calling the NotAllowedProcessor

```
onException(Exception.class) ③
```

③

onException to handle all other kind of Exceptions

```
.handled(true)  
.process(new GeneralErrorProcessor()); ④
```

④

By calling the GeneralErrorProcessor

```
public class NotAllowedProcessor implements Processor {  
  
    public void process(Exchange exchange) throws Exception {  
        ...  
    }  
}
```

```
throw new NullPointerException("null"); 5
```

5

Some code causes a NullPointerException

```
}
```

```
public void GeneralErrorProcessor implement Processor {  
    public void process(Exchange exchange) throws Exception {  
        ...  
        throw new AuthorizationException("forbidden"); 6
```

6

Some code causes a AuthorizationException

```
}
```

The first `onException` **1** is configured to handle all `AuthorizationException` exceptions by calling `NotAllowedProcessor` **2**. The second `onException` **3** is set up to handle all other kinds of exceptions by calling `GeneralErrorProcessor` **4**.

Because the `onExceptions` are calling processors, what could happen if these processors cause new exceptions to be thrown? The code in [listing 11.12](#) has been purposely set up to trigger such situations. Suppose at first `AuthorizationException` is thrown during routing, which leads to `NotAllowedProcessor` to be called, and unfortunately this processor throws `NullPointerException` **5**, which causes `onException` **3** to react and call `GeneralErrorProcessor`, which also unfortunately throws an `AuthorizationException` exception **6**. This scenario would lead to an endless circular error handling.

To avoid these unfortunate scenarios, Camel doesn't allow further error handling while already handling an error. In other words, when `NullPointerException` **5** is thrown the first time,

Camel detects that another exception was thrown during error handling, and prevents any further action from taking place. This is done by the

`org.apache.camel.processor.FatalFallbackErrorHandler`, which catches the new exception, logs a warning, sets this as the exception on the Exchange, and stops any further routing.

The following code shows a snippet of the warning logged by Camel:

```
2017-03-14 12:46:54,885 [main] ERROR
FatalFallbackErrorHandler-
Exception occurred while trying to handle previously thrown
exception on
exchangeId: ID-davsclaus-pro-57045-1426333614411-0-2 using
[Channel[DelegateSync[camelinaction.NotAllowedProcessor@3b9
38003]]].
The previous and the new exception will be logged in the
following.
2017-03-14 12:46:54,886 [main] ERROR
FatalFallbackErrorHandler
\--> Previous exception on exchangeId:
    ID-davsclaus-pro-57045-1426333614411-0-2
    camelinaction.AuthorizationException: Forbidden
    at
    camelinaction.NewExceptionTest$1.configure(NewExceptionTest
    .java:41)
2017-03-14 12:46:54,894 [main] ERROR
FatalFallbackErrorHandler
\--> New exception on exchangeId:
    ID-davsclaus-pro-57045-1426333614411-0-2
    java.lang.NullPointerException
    at
    camelinaction.NotAllowedProcessor.process(NotAllowedProcess
or.java:28)
```

You can see this in action by running the following Maven goal from the chapter 11/newexception directory:

```
mvn test -Dtest=NewExceptionTest
mvn test -Dtest=SpringNewExceptionTest
```

Instead of handling exceptions, in some situations you'll want to ignore the exception and continue routing, although those situations are rare.

11.4.6 IGNORING EXCEPTIONS

In section 11.4.3, you learned about how `onException` can handle exceptions. Handling an exception means that Camel will break out of the route. But sometimes all you want is to catch the exception and continue routing. You can do that in Camel using `continued`. All you have to do is to use `continued(true)` instead of `handled(true)`.

Suppose you want to ignore any `ValidationException` that may be thrown in the route, laid out in [listing 11.8](#). The following listing shows how to do this.

[Listing 11.13](#) Using continued to ignore validationExceptions

```
onException(JMSEception.class)
    .handled(true)
    .process(new GenerateFailureResponse());
onException(ValidationException.class)
    .continued(true); ①
```

①

Ignores all ValidationExceptions

```
from("netty4:tcp://0.0.0.0:4444?textline=true")
    .process(new ValidateOrderId())
    .to("jms:queue:order.status")
    .process(new GenerateResponse());
```

As you can see, all you have to do is add another `onException` that uses `continued(true)` ①.

NOTE You can't use both `handled` and `continued` on the same `onException`.

Now imagine that a message once again arrives at the TCP endpoint, and the Camel application routes the message, but this

time the validate processor throws `ValidationException`. This situation is illustrated in [figure 11.6](#).

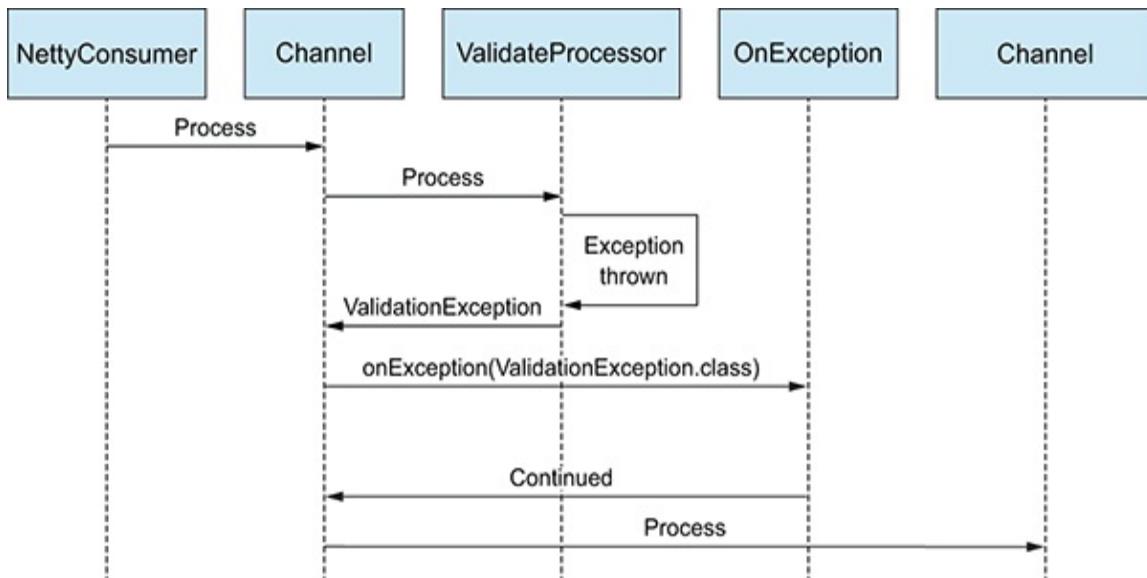


Figure 11.6 Sequence diagram of a message being routed and `ValidationException` being thrown from `ValidateProcessor`. The exception is handled and continued by the `onException` policy, causing the message to continue being routed as if the exception weren't thrown.

When `ValidateProcessor` throws `ValidationException`, it's propagated back to the channel, which lets the error handler kick in. The route has an `onException` defined that instructs the channel to continue routing the message—`continued(true)`. When the message arrives at the next channel, it's as if the exception weren't thrown. This is much different from what you saw in section 11.4.3 when using `handled(true)`, which causes the processing to break out and *not* continue routing.

You've learned a bunch of new stuff, so let's continue with the error-handler example and put your knowledge into practice.

11.4.7 IMPLEMENTING AN ERROR-HANDLER SOLUTION

Suppose your boss brings you a new problem. This time, the remote HTTP server used for uploading files is unreliable, and he wants you to implement a secondary failover to transfer the files by FTP to a remote FTP server.

You've been studying *Camel in Action*, and you've learned that Camel has extensive support for error handling and that you could use `onException` to provide this kind of feature. With great confidence, you fire up the editor and alter the route as shown in the following listing.

Listing 11.14 Route using error handling with failover to FTP

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(5).redeliveryDelay(10000));

onException(IOException.class).maximumRedeliveries(3) ❶
```

❶

Exception policy

```
.handled(true)
.to("ftp://gear@ftp.rider.com?password=secret");

from("file:/rider/files/upload?delay=15m")
.to("http://rider.com/upload?
user=gear&password=secret");
```

This listing adds `onException` ❶ to the route, telling Camel that in the case of `IOException`, it should try redelivering up to three times, using a 10-second delay. If there's still an error after the redelivery attempts, Camel will handle the exception and reroute the message to the FTP endpoint instead. The power and flexibility of the Camel routing engine shines here. `onException` is just another route, and Camel will continue on this route instead of the original route.

NOTE In [listing 11.14](#), it's only when `onException` is exhausted that it will reroute the message to the FTP endpoint ❶. The `onException` has been configured to redeliver up to three times before giving up and being exhausted.

The book's source code contains this example in the chapter11/usecase directory, and you can try it out yourself. The example contains a server and a client that you can start by using Maven:

```
mvn compile exec:java -PServer  
mvn compile exec:java -PClient
```

Both the server and client output instructions on the console about what to do next, such as copying a file to the target/rider folder to get the ball rolling.

Here, toward the end of this chapter, we've saved a great example for you to try. In this example, you'll use an external database, and then try to see what happens when you cut the connection between Camel and the database. To make the example easier to run, the book's accompanying source code uses a Docker image to use Postgres as the database.

11.4.8 BRIDGING THE CONSUMER WITH CAMEL'S ERROR HANDLER

At the start of this chapter, we discussed how Camel's error handler works. You learned that Camel's error handler takes effect during routing exchanges. If an error happens within the consumer prior to an exchange being created, the error is handled individually by each Camel component. Most of the components will log the error and not create any exchange to be routed.

To better understand the content of this section, take a look at [figure 11.7](#) (which is a copy of [figure 11.2](#)).

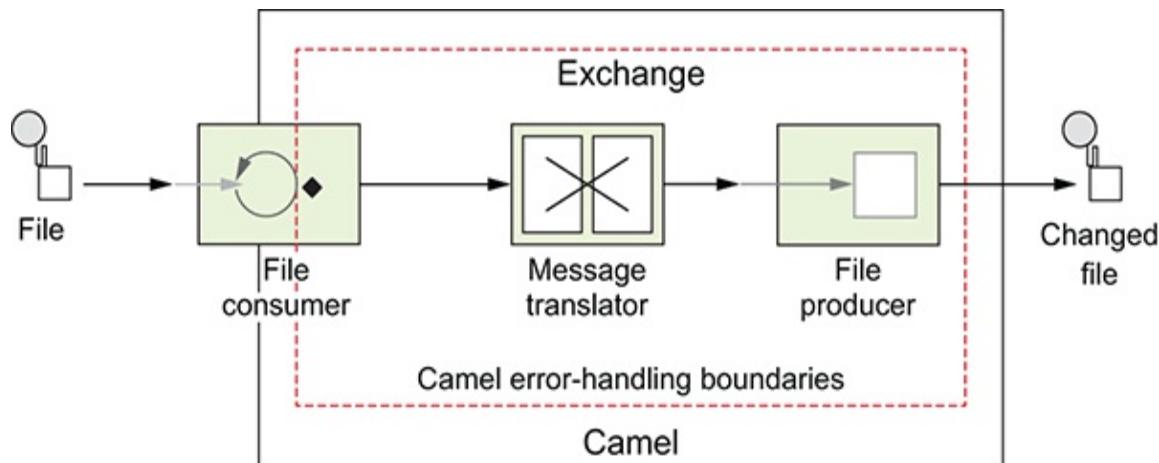


Figure 11.7 Camel’s error handling applies only within the lifecycle of an exchange. The error-handler boundaries are represented by the dashed square in the figure.

Earlier in the chapter we asked, “What happens if the file consumer can’t read the file?” You learned that this was component specific and that the file consumer would log a `WARN`, skip the file, and attempt to read the file again on the next poll. But what if you want to handle that error? What can you do? You can configure the consumer to bridge with Camel’s error handler. When an error is thrown internally in the consumer, and the bridge is enabled, the consumer creates an empty exchange with the caused error as the exception. The exchange is then routed by Camel, and as the routing engine detects the exception immediately, it allows the regular error handler to deal with the exception. This is illustrated in [figure 11.8](#).

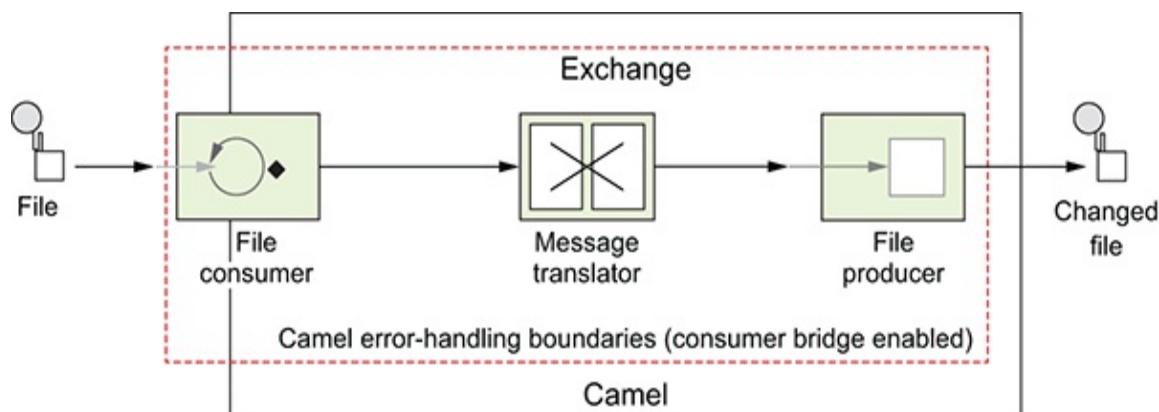


Figure 11.8 Camel’s error handling now includes early errors happening from within the consumer, when the bridge error-handler option has been enabled.

Notice that Camel's error-handling boundaries have expanded to include the consumer. If the consumer captures an exception earlier while attempting to detect or read incoming files, the exception is captured and sent along to Camel's error handler to deal with the exception. This functionality is known as *bridging the consumer* with Camel's error handler.

To learn how all this works, let's use an example to cover this from head to tail: a book-order system. A database is used to store incoming orders in a table. Then a Camel application polls the database table for any rows inserted into the table, and for each row a log message is printed. If an error occurs, Camel's error handler is configured to react and route a message to a dead letter channel. [Figure 11.9](#) presents this example in a visual style.

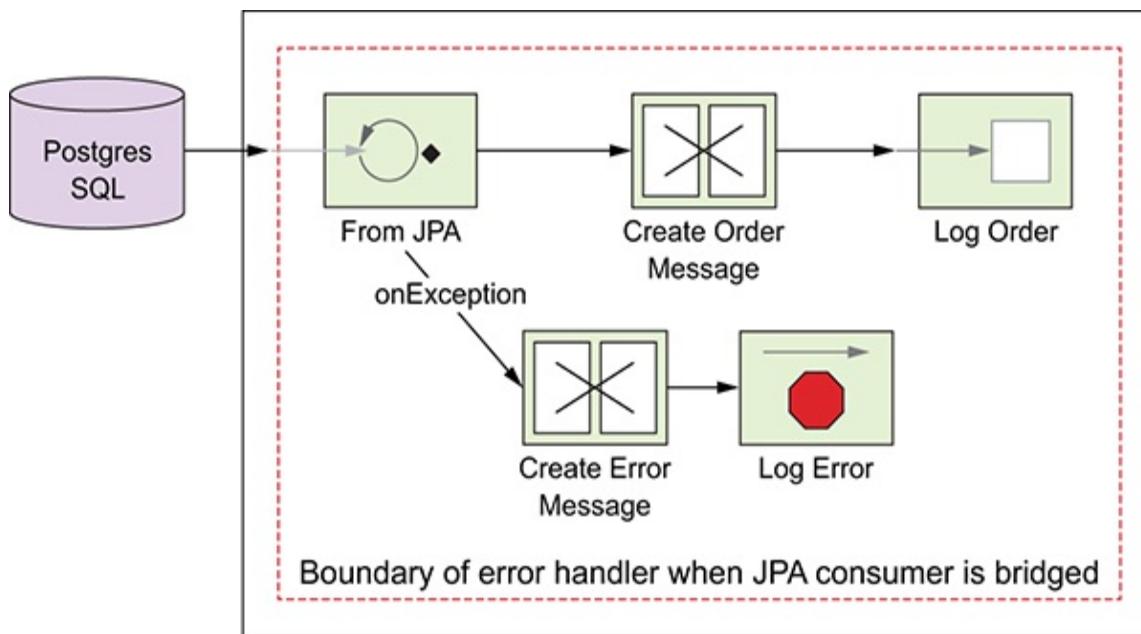


Figure 11.9 The JPA consumer is bridged to Camel's error handler so `onException` now also triggers when an error happens internally in the JPA consumer, such as no connection to the PostgreSQL database.

The book's source code contains this example in the `chapter11/bridge` directory. The example uses Docker to run the Postgres database, and instructions are provided in the `readme.md` file from the example source code.

If you have Docker set up correctly on your computer, the

database can be started in one line:

```
docker run --name my-postgres -p 5432:5432 -e  
POSTGRES_PASSWORD=secret -d postgres
```

When the Postgres database is up and running, you can run the Camel application using the following:

```
mvn clean install exec:java
```

The example is constructed so that at one point it waits for you to press Enter before continuing. The idea is that it gives you time to stop the Postgres database so you can see what happens in the Camel application when the consumer can't connect to the database, and therefore an error happens internally in the consumer. When you do this, the console should print something similar to this:

```
2017-05-11 20:08:39,425 [ction.BookOrder] WARN books - We  
do not care Connection to 0.0.0.0:5432 refused. Check that  
the hostname and port are correct and that the postmaster  
is accepting TCP/IP connections.
```

The preceding error message (highlighted in bold) tells you that an exception happened, but you don't care. The cause of the error follows with Connection to 0.0.0.0:5432 refused. That last part of the error message is a JPA error message when a connection to the database can't be established. (The Postgres database runs on host 0.0.0.0.)

If you take a moment to dive into the source code of this example, you'll learn that this error message was created by Camel's error handler, as shown in the following listing.

[Listing 11.15](#) Example that bridges the consumer with Camel's error handler

```
<camelContext id="camel"  
xmlns="http://camel.apache.org/schema/spring">  
  
    <endpoint id="newBookOrders"  
uri="jpa:camelinaction.BookOrder">
```

1

Configures the JPA endpoint outside the route, allowing you to configure each option using <property> elements

```
<property key="delay" value="1000"/>  
<property key="bridgeErrorHandler" value="true"/> 2
```

2

Bridges the consumer with Camel's error handler

```
<property key="backoffMultiplier" value="10"/> 3
```

3

Turns on back-off to multiply the polling delay with x10 after one error

```
<property key="backoffErrorThreshold" value="1"/>  
</endpoint>  
  
<onException>  
  <exception>java.lang.Exception</exception>  
  <redeliveryPolicy  
    logExhaustedMessageHistory="false" 4
```

4

Tones down logging noise when Camel's error handler is handling the exception

```
    logExhausted="false"/>  
  <handled>  
    <constant>true</constant>  
  </handled>  
  <log message="We do not care ${exception.message}"  
    loggingLevel="WARN"/> 5
```

5

Prints a logging message indicating you don't care about the exception

```
</onException>  
  
<route id="books">
```

```
<from uri="ref:newBookOrders"/>
```

6

6

The route that polls the Postgres database

```
<log message="Order ${body.orderId} -  
${body.title}" />  
</route>  
</camelContext>
```

To configure the JPA consumer, you set up the endpoint ❶ and provide it with the ID newBookOrders. Each option is configured using a `<property>` tag. Notice that you can also configure the endpoint using the more common URI style. To bridge the consumer with Camel's error handler, all you have to do is set `bridgeErrorHandler` to `true` ❷. In case of an error, you don't want the consumer to continue polling as frequently as normal, so you apply the back-off multiplier option with a value of 10 ❸. Instead of polling every 1,000 ms, the consumer will poll $10 \times 1,000 \text{ ms} = 10,000 \text{ ms}$ (1 sec → 10 sec). The back-off threshold tells Camel after how many subsequent errors the back-off should apply. In this example, that occurs after the first error. But if you set the threshold to 5, only after five subsequent errors will the polling be delayed by the back-off multiplier. When an exception happens, you want Camel's error handler to react and print only a brief custom message to the log ❹. To avoid having any additional information logged, you turn off the logging of the exhausted error on the error handler ❺; this ensures that only the custom message is being logged. And at the bottom of the listing is the route that uses the JPA consumer to pick up new book orders ❻.

TIP We encourage you to try this example on your own. For example, see what happens when you turn off `bridgeErrorHandler`. You could also try changing the back-off thresholds and see what happens if you remove the `onException` code.

The example uses a few options related to back-off; let's take a moment to look at these options formally in a table.

USING BACK-OFF TO LET THE CONSUMER BE LESS AGGRESSIVE

Table 11.6 lists all the back-off options that Camel provides out of the box.

Table 11.6 Back-off options to let a consumer be less aggressive during polling

Option	Description
bac kof fMu lti plier	Lets the polling consumer back off if there's been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt happens again. When this option is in use, <code>backoffIdleThreshold</code> and/or <code>backoffErrorThreshold</code> must also be configured.
bac kof fId leT hre sho ld	The number of subsequent idle polls that should happen before <code>backoffMultiplier</code> kicks in.
bac kof fEr ror Thr esh old	The number of subsequent error polls (failed due to an error) that should happen before <code>backoffMultiplier</code> kicks in.

Before finishing up this chapter, we need to take a look at a few more error-handling features. They're used rarely, but they provide power in situations where you need more fine-grained control.

11.5 Working with other error-handling features

Here are some of the other features Camel provides for error handling:

- `onWhen`—Allows you to dictate when an exception policy is in use
- `onExecutionOccurred`—Allows you to execute code *right* after an exception occurs
- `onRedeliver`—Allows you to execute code before the message is redelivered
- `retryWhile`—Allows you, at runtime, to determine whether to continue redelivery or to give up

We'll look at each in turn.

11.5.1 USING ONWHEN

The `onWhen` predicate filter allows more fine-grained control over when `onException` should be triggered.

Suppose a new problem has emerged with your application in [listing 11.14](#). This time, the HTTP service rejects the data and returns an HTTP 500 response with the constant text `ILLEGAL DATA`. Your boss wants you to handle this by moving the file to a special folder where it can be manually inspected to see why it was rejected.

First, you need to determine when an HTTP error 500 occurs and whether it contains the text `ILLEGAL DATA`. You decide to create a Java method that can test this, as shown in the following listing.

Listing 11.16 A helper to determine whether an HTTP error 500 occurred

```
public final class MyHttpUtil {
    public static boolean isIllegalDataError(
        HttpOperationFailedException cause) {
        int code = cause.getStatusCode();      ①
```

①

Gets HTTP status code

```

        if (code != 500) {
            return false;
        }
        return "ILLEGAL
DATA".equals(cause.getResponseBody().toString());
    }
}

```

When the HTTP operation isn't successful, the Camel HTTP component will throw an `org.apache.camel.component.http.HttpOperationFailedException` exception, containing information about why it failed. The `getStatusCode` method on `HttpOperationFailedException` ① returns the HTTP status code. This allows you to determine whether it's an HTTP error code 500 with the `ILLEGAL DATA` body text.

Now you need to set up Camel's error handler to use the bean from [listing 11.16](#) to determine whether it's this special error your boss told you to handle. You can do this using the `onwhen` predicate with `onException`:

```

onException(HttpOperationFailedException.class)
    .onWhen(bean(MyHttpUtil.class, "isIllegalData"))
    .handled(true)
    .to("file:/rider/files/illegal");

```

When the HTTP server fails, an `HttpOperationFailedException` exception is thrown, and `onException` is triggered. The `onwhen` predicate means that the `isIllegalData` method on the `MyHttpUtil` bean ([listing 11.16](#)) returns either `true` or `false`. If `true` was returned, the exception is handled, and Camel will store the message in the filesystem in the `rider/files/illegal` folder.

Here's how you use `onwhen` in XML DSL:

```

<onException>
    <exception>org.apache.camel.component.http.
        HttpOperationFailedException</exception>
    <onwhen><method ref="myHttpUtil" method="isIllegalData"/>
    </onwhen>
    <handled><constant>true</constant></handled>
    <to uri="file:/rider/files/illegal"/>
</onException>

```

```
<bean id="myHttpUtil" class="com.rider.MyHttpUtil"/>
```

onWhen is a general function that also exists in other Camel features, such as interceptors and onCompletion, so you can use this technique in various situations.

onExceptionOccurred, discussed next, allows you to call a Camel processor right after an exception is thrown.

11.5.2 USING ONEXCEPTIONOCCURRED

onExceptionOccurred is designed to allow code to be executed right after an exception is thrown, and before any redelivery may happen. This allows you to do some custom processing immediately after the exception is thrown, such as doing custom logging, or any need you may have.

onExceptionOccurred can be configured on the error handler:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .onExceptionOccurred(new MyLogExceptionProcessor()));
```

Or on onException:

```
onException(IOException.class)
    .maximumRedeliveries(5)
    .onExceptionOccurred(new MyLogExceptionProcessor());
```

In XML DSL, onExceptionOccurred is configured as a reference to a bean, as follows:

```
<onException onExceptionOccurredRef="myLogException">
    <exception>java.io.IOException</exception>
</onException>

<bean id="myLogException"
      class="com.mycompany.MyLogExceptionProcessor"/>
```

Next, let's look at onRedeliver, which is similar to onExceptionOccurred but instead of triggering right after the exception, it triggers right before any redelivery is about to take place.

11.5.3 USING ONREDELIVER

The purpose of `onRedeliver` is to allow code to be executed before a redelivery is performed. This gives you the power to do custom processing on the exchange before Camel makes a redelivery attempt. You can, for instance, use it to add custom headers to indicate to the receiver that this is a redelivery attempt. `onRedeliver` uses `org.apache.camel.Processor`, in which you implement the code to be executed.

`onRedeliver` can be configured on the error handler:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .onRedeliver(new MyOnRedeliveryProcessor()));
```

Or on `onException`:

```
onException(IOException.class)
    .maximumRedeliveries(5)
    .onRedeliver(new MyOtherOnRedeliveryProcessor());
```

In XML DSL, `onRedeliver` is configured as a reference to a bean, as follows:

```
<onException onRedeliveryRef="myOtherRedelivery">
    <exception>java.io.IOException</exception>
</onException>

<bean id="myOtherRedelivery"
    class="com.mycompany.MyOtherOnRedeliveryProcessor"/>
```

Finally, let's look at one last feature: `retryWhile`.

11.5.4 USING RETRYWHILE

`retryWhile` is used when you want fine-grained control over the number of redelivery attempts. It's also a predicate that's scoped, so you can define it on the error handler or on `onException`.

You can use `retryWhile` to implement your own generic retry rule set that determines how long it should retry. The following listing shows some skeleton code demonstrating how this can be done.

Listing 11.17 Skeleton code to illustrate principle of using retryWhile

```
public class MyRetryRuleset {  
    public boolean shouldRetry(1  
        @Header(Exchange.REDELIVERY_COUNTER)  
        Integer counter,  
        Exception causedBy) {  
        ...  
        // return true or false  
    }  
}
```

Using your own `MyRetryRuleset` class, you can implement your own logic to determine whether it should continue retrying. In [listing 11.17](#), the method is named `shouldRetry` **1** and returns a boolean. Camel has no restriction on the name of the method. For example, the method could've been named `anotherBrickInTheWall` or any other fantastic name that may spring to mind. If the method returns `true`, a redelivery attempt is conducted; if it returns `false`, it gives up.

To use your rule set, you configure `retryWhile` on `onException` as follows:

```
onException(IOException.class).retryWhile(bean(MyRetryRuleset.class));
```

In XML DSL, you configure `retryWhile` as shown:

```
<onException>  
    <exception>java.io.IOException</exception>  
    <retryWhile><method ref="myRetryRuleset"/></retryWhile>  
</onException>  
  
<bean id="myRetryRuleset"  
      class="com.mycompany.MyRetryRuleset"/>
```

That gives you fine-grained control over the number of redelivery attempts performed by Camel.

That's it! We've now covered all the features Camel provides for fine-grained control over error handling.

11.6 Summary and best practices

In this chapter, you saw how recoverable and irrecoverable errors are represented in Camel. We also looked at all the provided error handlers, focusing on the most important of them. You saw that Camel can control how exceptions are dealt with, using redelivery policies to set the scene and exception policies to handle specific exceptions differently. Finally, we looked at what Camel has to offer when it comes to fine-grained control over error handling, putting you in control of error handling in Camel.

Let's revisit some of the key ideas from this chapter, which you can apply to your own Camel applications:

- *Error handling is hard*—Realize from the beginning that the unexpected can happen and that dealing with errors is hard. The challenge keeps rising when businesses have more and more of their IT portfolio integrated and operate it 24/7/365.
- *Error handling isn't an afterthought*—When IT systems are being integrated, they exchange data according to agreed-upon protocols. Those protocols should also specify how errors will be dealt with.
- *Separate routing logic from error handling*—Camel allows you to separate routing logic from error-handling logic. This avoids cluttering up your logic, which otherwise could become harder to maintain. Use Camel features such as error handlers, `onException`, and `doTry ... doCatch`.
- *Try to recover*—Some errors are recoverable, such as connection errors. You should apply strategies to recover from these errors.

- *Capture error details*—When errors happen, try to capture details by ensuring that sufficient information is logged.
- *Use monitoring tooling*—Use tooling to monitor your Camel applications so the tooling can react and alert personnel if severe errors occur. Chapter 16 covers such strategies.
- *Build unit tests*—Build unit tests that simulate errors to see whether your error-handling strategies are up to the task. Chapter 9 covered testing.
- *Bridge the consumer with Camel's error handler*—This allows you to deal with these errors in the same way as errors happened during routing.

This chapter covered the Dead Letter Channel EIP (among others), which deals with errors by moving failure messages to a dead letter queue for later inspection. The next chapter covers how Camel works with transactions and how to set up and use these transactions in various containers such as Java EE, Spring, and OSGi. The chapter also talks about what can be done in the absence of transactions by using compensation and idempotency.

12

Transactions and idempotency

This chapter covers

- Understanding why you need transactions
- Using and configuring transactions
- Understanding the differences between local and global transactions
- Using transactions with messaging and databases
- Rolling back transactions
- Compensating when transactions aren't supported
- Preventing duplicate messages by using idempotency
- Learning about the idempotent repository implementations shipped out of the box

To help explain transactions, let's look at an example from real life. You may well have ordered this book from Manning's online bookstore, and if you did, you likely followed these steps:

1. Find the book *Camel in Action, 2nd Edition*.
2. Put the book into the basket.
3. Maybe continue shopping and look for other books.
4. Go to the checkout.

5. Enter shipping and credit card details.
6. Confirm the purchase.
7. Wait for the confirmation.
8. Leave the web store.

What seems like an everyday scenario is a fairly complex series of events. You have to put books in the basket before you can check out; you must fill in the shipping and credit card details before you can confirm the purchase; if your credit card is declined, the purchase won't be confirmed; and so on. The ultimate resolution of this transaction is one of two states: either the purchase is accepted and confirmed, or the purchase is declined, leaving your credit card balance uncharged.

This particular story involves computer systems because it's about using an online bookstore, but the same main points happen when you shop in the supermarket. Either you leave the supermarket with your groceries or without.

In the software world, transactions are often explained in the context of SQL statements manipulating database tables—updating or inserting data. While the transaction is in progress, a system failure could occur, and that would leave the transaction's participants in an inconsistent state. That's why the series of events is described as *atomic*: either they all are completed or they all fail—it's all or nothing. In transactional terms, they either *commit* or *roll back*.

NOTE You probably know about the database ACID properties, so we won't explain what *atomic*, *consistent*, *isolated*, and *durable* mean in the context of transactions. If you aren't familiar with ACID, the Wikipedia page is a good place to start learning about it: <http://en.wikipedia.org/wiki/ACID>.

In this chapter, we'll first look at the reasons why you should use transactions (in the context of Rider Auto Parts). Then we'll look

at transactions in more detail and at Spring’s transaction management, which orchestrates the transactions. You’ll learn about the difference between local and global transactions and how to configure and use transactions. In the middle of the chapter, you’ll see how to compensate when you’re using resources that don’t support transactions. The last part of the chapter covers *idempotency*, which deals with handling duplicate messages. This can often happen when transactions can’t be used, and data exchange between distributed systems may have to replay messages to deal with an unreliable network between the systems. The last section of the chapter covers a common use-case of clustering an application and using clustered idempotency to coordinate work between the nodes, so each node doesn’t process duplicate messages.

12.1 Why use transactions?

Using transactions makes sense for many reasons. But before focusing on using transactions with Camel, let’s look at what can go wrong when you *don’t* use transactions.

In this section, you’ll review an application that Rider Auto Parts uses to collect metrics that will be published to an incident management system. You’ll see what goes wrong when the application doesn’t use transactions, and then you’ll apply transactions to the application.

12.1.1 THE RIDER AUTO PARTS PARTNER INTEGRATION APPLICATION

Lately, Rider Auto Parts has had a dispute with a partner about whether its service meets the terms of the service-level agreement (SLA). When such incidents occur, it’s often a labor-intensive task to investigate and remedy the incident.

In light of this, Rider Auto Parts has developed an application to record what happens, as evidence for when a dispute comes up. The application periodically measures the communication between Rider Auto Parts and its external partner servers. The

application records performance and uptime metrics, which are sent to a JMS queue, where the data awaits further processing.

Rider Auto Parts already has an existing incident management application with a web-user interface for upper management. What's missing is an application to populate the collected metrics to the database used by the incident management application. [Figure 12.1](#) illustrates the scenario.

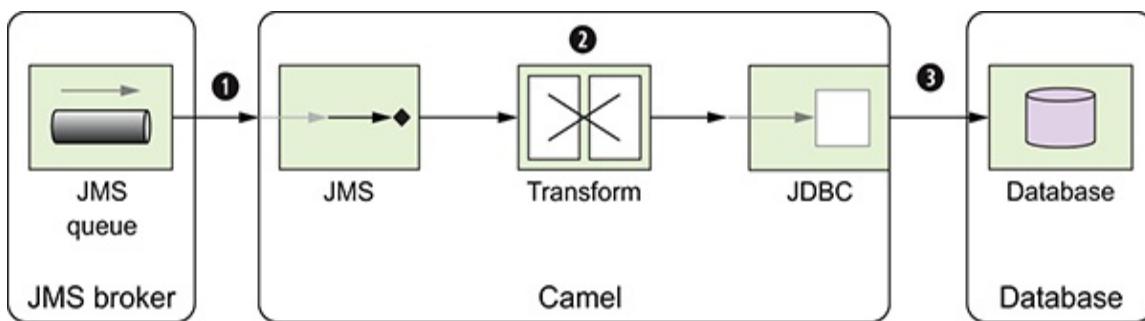


Figure 12.1 Partner reports are received from the JMS broker, transformed in Camel to SQL format, and then written to the database.

It's a fairly simple task: a JMS consumer listens for new messages on the JMS queue ①. Then the data is transformed from XML to SQL ② before it's written to the database ③.

In no time, you can come up with a route that matches [figure 12.1](#):

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

<propertyPlaceholder id="properties"
location="camelinaction/sql.properties"/>

<route id="partnerToDB">
    <from uri="activemq:queue:partners"/>
    <bean ref="partner" method="toMap"/>
    <to uri="sql:{sql-insert}?dataSource=#myDataSource"/>
</route>
</camelContext>
```

The reports are sent to the JMS queue in a simple in-house XML format, like this:

```
<?xml version="1.0"?>
<partner id="123">
    <date>201702250815</date>
    <code>200</code>
    <time>3921</time>
</partner>
```

The database table that stores the data is also mapped easily because it has the following layout:

```
create table partner_metric
( partner_id varchar(10), time_occurred varchar(20),
  status_code varchar(3), perf_time varchar(10) )
```

That leaves you with the fairly simple task of mapping the XML to the database. Because you're pragmatic and want to make a simple and elegant solution that anybody should be capable of maintaining in the future, you decide not to bring in the big guns with the Java Persistence API (JPA) or Hibernate. You put the mapping code shown in the following listing in a good old-fashioned bean.

Listing 12.1 Using a bean to map from XML to SQL

```
import org.apache.camel.language.XPath;

public class PartnerServiceBean {

    public Map toMap(@XPath("partner/@id") int id, ①
①
        Extracts data from XML payload
        @XPath("partner/date/text()") String
        date,
        @XPath("partner/code/text()") int
        statusCode,
        @XPath("partner/time/text()") long
        responseTime) {
        Map map = new HashMap(); ②
②
```

Constructs Map with values to insert using SQL

```
    map.put("id", id);
    map.put("date", date);
    map.put("code", statusCode);
    map.put("time", responseTime);
    return map;
}
}
```

Coding the mapping logic that extracts the data from XML to a Map in these 10 or so lines was faster than getting started on the JPA wagon or opening any heavyweight and proprietary mapping software.

First, you define the method to accept the four values to be mapped. Notice that you use the @XPath annotation to grab the data from the XML document ①. Then you use Map to store the input values ②. The SQL INSERT statement is externalized from Java code and defined in a properties file, which you name sql.properties:

```
sql-insert=insert into partner_metric (partner_id,
time_occurred,
status_code, perf_time) values (:#id, :#date, :#code,
:#time)
```

Notice that the keys from the Map ② match the values in the SQL statement (for example, :#id matching the ID, and :#date matching the date). Using :#key is how the Camel SQL component supports mapping from the message body or headers to the SQL dynamic placeholders.

NOTE In the first edition of this book, we used the Camel JDBC component for this example. This time we're using the SQL component, because it's been improved. For example, the SQL component uses prepared statements, whereas JDBC uses regular statements; this should yield better performance on the database. We, the authors, also admire the camel-elsql component that allows you to use a lightweight DSL to define

the SQL queries.

To test this, you can crank up a unit test as follows:

```
public void testSendPartnerReportIntoDatabase() throws  
Exception {  
    String sql = "select count(*) from partner_metric";  
    assertEquals(0, jdbc.queryForInt(sql)); 1
```

1

Asserts there are no rows in database

```
    String xml = "<?xml version=\"1.0\"?>  
                + <partner id=\"123\">  
<date>201702250815</date>  
                + <code>200</code><time>4387</time>  
</partner>";  
    template.sendBody("activemq:queue:partners", xml);  
    Thread.sleep(5000);  
    assertEquals(1, jdbc.queryForInt(sql)); 2
```

2

Asserts one row was inserted into database

}

This test method outlines the principle. First you check that the database is empty **1**. Then you construct sample XML data and send it to the JMS queue using the Camel ProducerTemplate. Because the processing of the JMS message is asynchronous, you must wait a bit to let it process (the book's source code uses NotifyBuilder to wait instead of Thread.sleep). At the end, you check that the database contains one row **2**.

12.1.2 SETTING UP THE JMS BROKER AND THE DATABASE

To run this test, you need to use a local JMS broker and a database. You can use Apache ActiveMQ as the JMS broker and Apache Derby as the database. Derby can be used as an in-

memory database without the need to run it separately. ActiveMQ is an extremely versatile broker, and it's even embeddable in unit tests.

All you have to do is master a bit of Spring XML magic to set up the JMS broker and the database, as shown in the following listing.

Listing 12.2 XML configuration for the Camel route, JMS broker, and database

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

    <bean id="partner"
          class="camelinaction.PartnerServiceBean"/>

    <camelContext id="camel"
                  xmlns="http://camel.apache.org/schema/spring">
        <propertyPlaceholder id="properties"
            location="camelinaction/sql.properties"/>
        <route id="partnerToDB">
            <from uri="activemq:queue:partners"/>
            <bean ref="partner" method="toMap"/>
            <to uri="sql:{{sql-insert}}?dataSource=#myDataSource"/>
        </route>
    </camelContext>

    <bean id="activemq" ①
```

①

Configures ActiveMQ component

```
class="org.apache.activemq.camel.component.ActiveMQComponen  
t">  
    <property name="brokerURL"  
value="tcp://localhost:61616"/>  
</bean>  
  
<broker:broker useJmx="false" persistent="false"  
brokerName="localhost">  
    <broker:transportConnectors> 2
```

2

Sets up embedded JMS broker

```
<broker:transportConnector  
uri="tcp://localhost:61616"/>  
    </broker:transportConnectors>  
</broker:broker>  
  
<bean id="myDataSource"  
class="org.springframework.jdbc.datasource.DriverManagerDat  
aSource"> 3
```

3

Sets up database

```
<property name="driverClassName"  
value="org.apache.derby.jdbc.EmbeddedXADataSource"/>  
    <property name="url"  
value="jdbc:derby:memory:order;create=true"/>  
</bean>  
  
</beans>
```

In [listing 12.2](#), you first define the partner bean from [listing 12.1](#) as a Spring bean that you'll use in the route. Then, to allow Camel to connect to ActiveMQ, you must define it as a Camel component **1**. The brokerURL property is configured with the URL for the remote ActiveMQ broker, which, in this example, happens to be running on the same machine. Then you set up a local embedded ActiveMQ broker **2**, which is configured to use

TCP connectors. Finally, you set up the JDBC data source ③.

Using VM instead of TCP with an embedded ActiveMQ broker

If you use an embedded ActiveMQ broker, you can use the VM protocol instead of TCP; doing so bypasses the entire TCP stack and is much faster. For example, in [listing 12.2](#), you could use `vm://localhost` instead of `tcp://localhost:61616`. The `localhost` in `vm://localhost` is the broker name, not a network address. For example, you could use `vm://myCoolBroker` as the broker name and configure the name on the broker tag accordingly: `brokerName="myCoolBroker"`.

A plausible reason that you're using `vm://localhost` in [listing 12.2](#) is that the engineers are lazy, and they changed the protocol from TCP to VM but left the broker name as `localhost`.

Embedding an ActiveMQ broker is a good use-case for unit and integration tests, and for some application servers that provide messaging out of the box such as Apache ServiceMix and JBoss Fuse. For other use-cases, it's recommended to run ActiveMQ standalone, which allows you to separate the brokers from your applications. ServiceMix and JBoss Fuse users may consider an architecture in which they configure some nodes to be dedicated ActiveMQ brokers and other nodes to host their applications. This allows you to cleanly separate brokers from applications, which is a recommended practice. But the one-size-fits-all rule applies here; in some use-cases, colocating the ActiveMQ brokers with your applications can be good practice (for example, if the messaging load is light, and you don't need to scale the brokers independently from your applications).

The full source code for this example is located in the chapter12/riderautoparts-partner directory, and you can try out the example by running the following Maven goal:

```
mvn test -Dtest=RiderAutoPartsPartnerTest
```

In the source code, you'll also see how we prepared the database by creating the table and dropping it after testing.

12.1.3 THE STORY OF THE LOST MESSAGE

The previous test is testing a positive situation, but what happens if the connection to the database fails? How can you test that?

Chapter 9 covered how to simulate a connection failure using Camel interceptors. Writing a unit test is just a matter of putting all that logic in a single method, as shown in the following listing.

Listing 12.3 Simulating a connection failure that causes lost messages

```
public void testNoConnectionToDatabase() throws Exception {  
    NotifyBuilder notify = new  
    NotifyBuilder(context).whenDone(1).create();  
  
    RouteBuilder rb = new RouteBuilder() { 1
```

1

Simulates no connection to database

```
        public void configure() throws Exception {  
            interceptSendToEndpoint("sql:*")  
                .throwException(new  
ConnectException("Cannot connect"));  
        }  
    };  
  
    RouteDefinition route =  
context.getRouteDefinition("partnerToDB");  
    route.adviceWith(context, rb); 2
```

2

Advises simulation into existing route

```
String sql = "select count(*) from partner_metric";
assertEquals(0, jdbc.queryForInt(sql)); 3
```

3

SQL to select number of rows in the database

```
String xml = "<?xml version=\"1.0\"?>
    + <partner id=\"123\">
<date>201611150815</date>
    + <code>200</code><time>4387</time>
</partner>";

template.sendBody("activemq:queue:partners", xml);
assertTrue(notify.matches(10, TimeUnit.SECONDS));
assertEquals(0, jdbc.queryForInt(sql)); 4
```

4

Asserts no rows inserted into database

```
}
```

To test a failed connection to the database, you need to intercept the routing to the database and simulate the error. You do this with `RouteBuilder`, where you define this scenario ①. Next you need to add the interceptor with the existing route ②, which is done using the `adviceWith` method. The remainder of the code is almost identical to the previous test, but you test that no rows are added to the database, as the `select count(*)` ③ SQL query should return 0 rows ④.

NOTE You can read about simulating errors by using interceptors in chapter 9, section 9.4.3.

The test runs successfully. But what happened to the message you sent to the JMS queue? It wasn't stored in the database, so where did it go?

It turns out that the message is lost because you're not using transactions. By default, the JMS consumer uses *auto-acknowledge mode*: the client acknowledges the message when it's received, and the message is dequeued from the JMS broker.

What you must do instead is use *transacted-acknowledge mode*. We'll look at how to do this in section 12.3, but first we'll discuss how transactions work in Camel.

NOTE The JMS specification also defines a *client-acknowledge mode*. This mode isn't often used with Camel, as you (the client) need to call the acknowledge method on the JMS message to mark the message as successfully consumed; performing this action requires you to write Java code. The book's source code contains an example of using JMS client-acknowledge mode in the chapter12/riderautoparts-partner directory. You can try this example using the following Maven goal:

```
mvn test -Dtest=RiderAutoPartsPartnerClientAcknowledgeModeTest
```

12.2 Transaction basics

A *transaction* is a series of events. The start of a transaction is often named `begin`, and the end is `commit` (or `rollback` if the transaction isn't successfully completed). [Figure 12.2](#) illustrates this.

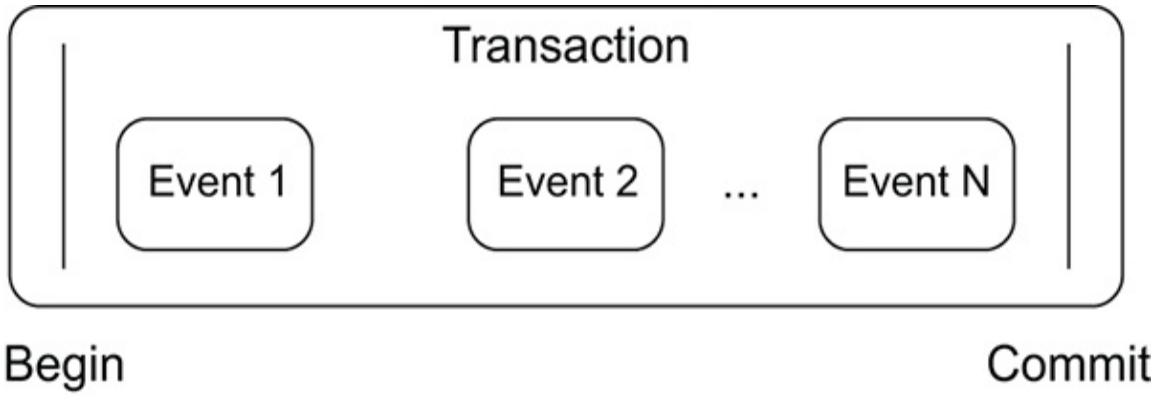


Figure 12.2 A transaction is a series of events between `begin` and `commit`.

To demonstrate the sequence in [figure 12.2](#), you could write *locally managed transactions* (the transaction is managed manually in the code). The following code illustrates this:

```
TransactionManager tm = ...
Transaction tx = tm.getTransaction();
try {
    tx.begin();
    ...
    tx.commit();
} catch (Exception e) {
    tx.rollback();
}
```

You start the transaction using the `begin` method. Then you have a series of events to do whatever work needs to be done. At the end, you either commit or roll back the transaction, depending on whether an exception is thrown.

You may already be familiar with this principle, and transactions in Camel use the same principle at a higher level of abstraction. In Camel transactions, you don't invoke `begin` and `commit` methods from Java code; you use declarative transactions, which can be configured using Java code or in XML files. Camel doesn't reinvent the wheel and implement a transaction manager, which is a complicated piece of technology to build. Instead, Camel uses Spring's transaction support.

NOTE For more information on Spring's transaction

management, see chapter 10, “Transaction Management,” in the *Spring Framework Reference Documentation*:<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/transaction.html>.

Now that we’ve established that Camel works with Spring’s transaction support, let’s look at how they work together.

12.2.1 UNDERSTANDING SPRING’S TRANSACTION SUPPORT

To understand how Camel works together with Spring’s transaction support, take a look at [figure 12.3](#). This figure shows that Spring orchestrates the transaction while Camel takes care of the rest.

Spring TransactionManager API vs. implementations

Camel uses Spring Transaction to manage transactions via its TransactionManager API. Depending on the kinds of resources that are taking part in the transaction, an appropriate implementation of the transaction manager must be chosen. Spring offers a number of transaction managers out of the box that work for various local transactions such as JMS and JDBC. But for global transactions, you must use a third-party JTA transaction manager implementation; JTA transaction manager is provided by Java EE application servers. Spring doesn’t offer that out of the box, only the necessary API abstract that Camel uses. We cover using third-party transaction managers in section 12.3.2.

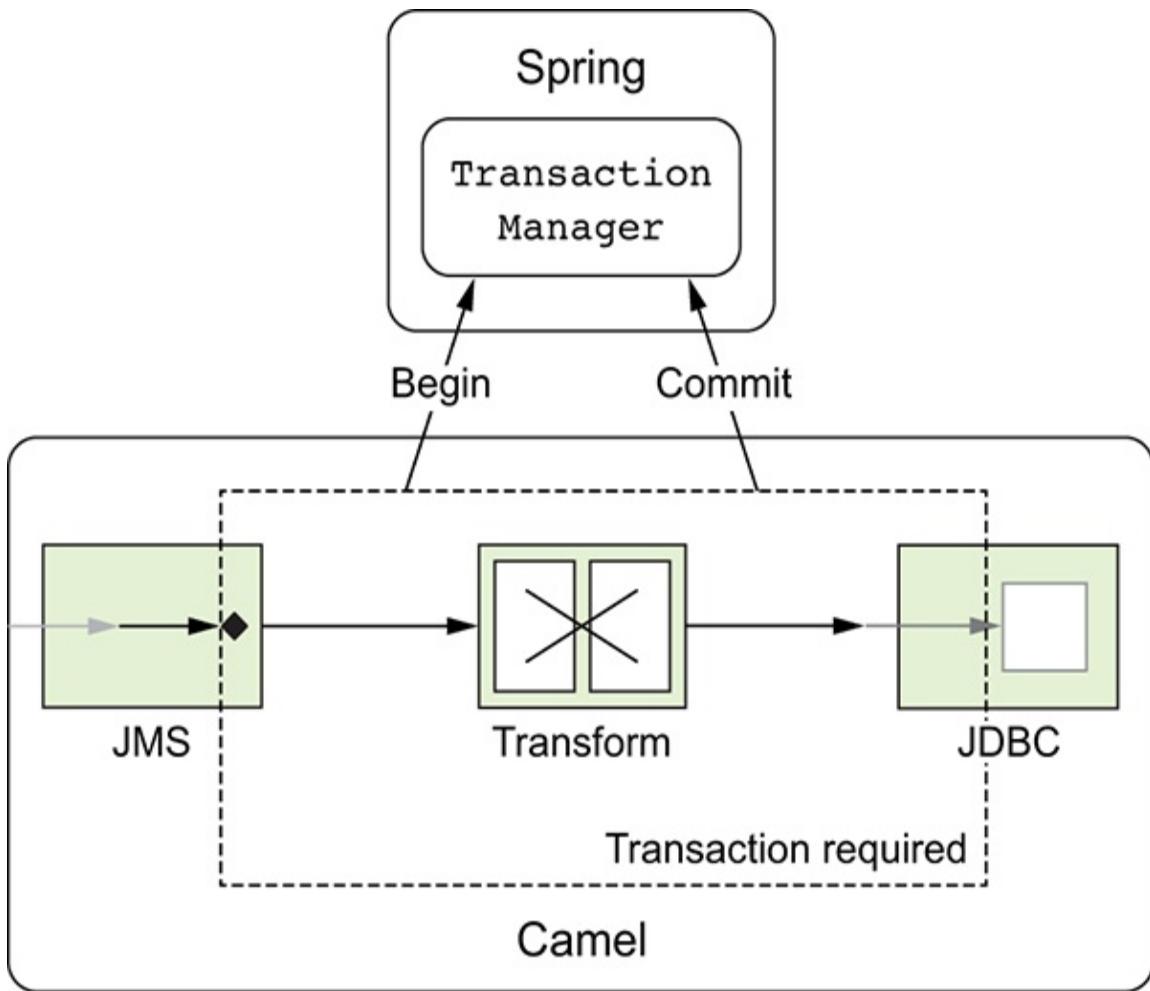


Figure 12.3 Spring's `TransactionManager` orchestrates the transaction by issuing `beginS` and `commits`. The entire Camel route is transacted, and the transaction is handled by Spring.

Figure 12.4 adds more details, to illustrate that the JMS broker also plays an active part in the transaction. You can see how the JMS broker, Camel, and the Spring `JmsTransactionManager` work together. `JmsTransactionManager` orchestrates the resources that participate in the transaction ①, which in this example is the JMS broker.

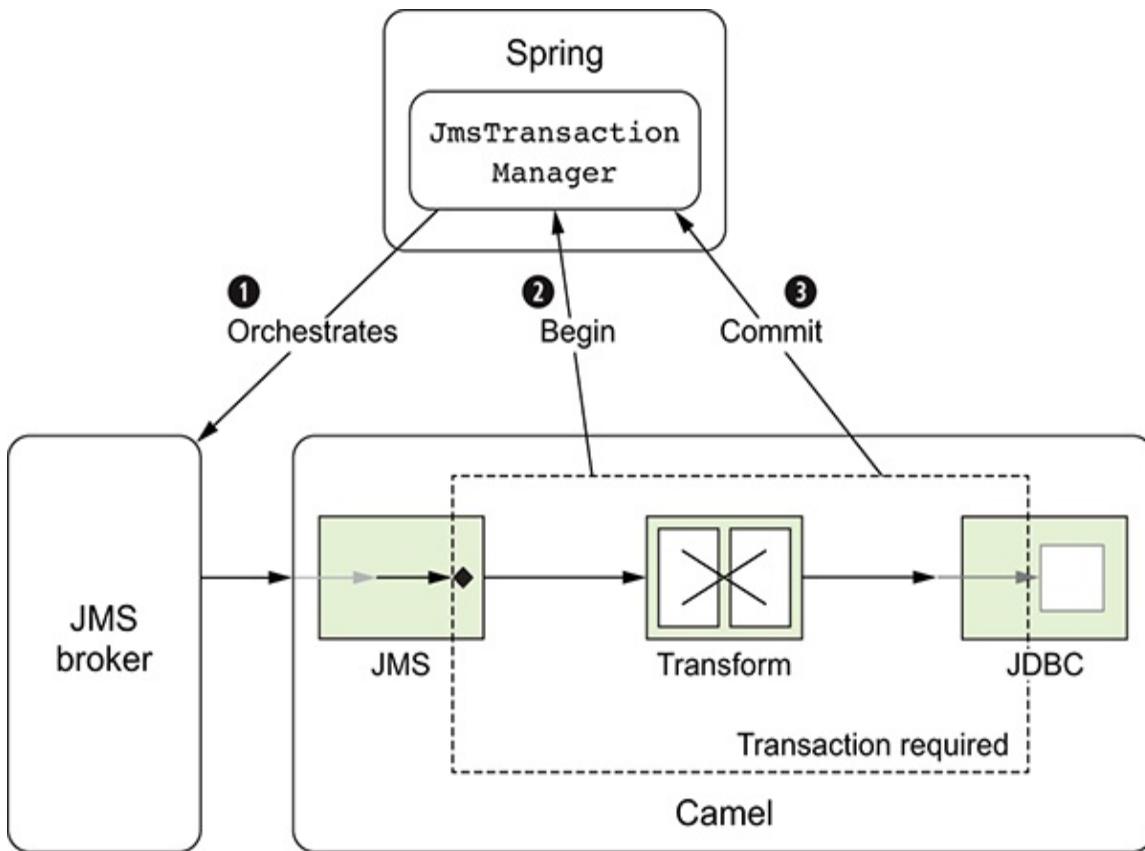


Figure 12.4 The Spring `JmsTransactionManager` orchestrates the transaction with the JMS broker. The Camel route completes successfully and signals the commit to the `JmsTransactionManager`.

When a message has been consumed from the JMS queue and fed into the Camel application, Camel issues a begin ② to `JmsTransactionManager`. Depending on whether the Camel route completes with success or failure ③, `JmsTransactionManager` will ensure that the JMS broker commits or rolls back.

It's now time to see how this works in practice. In the next section, you'll fix the lost-message problem by adding transactions.

12.2.2 ADDING TRANSACTIONS

At the end of section 12.1, you left Rider Auto Parts with the problem of losing messages because you didn't use transactions. Your task now is to apply transactions, which should remedy the problem.

You'll start by introducing Spring transactions to the XML file and adjusting the configuration accordingly. The following listing shows how this is done.

Listing 12.4 XML configuration using Spring transactions

```
<bean id="activemq"
      class="org.apache.activemq.camel.component.ActiveMQComponen
      t">
    <property name="transacted" value="true"/> 1
```

1
Enables transacted-acknowledge mode

```
    <property name="transactionManager" ref="txManager"/>
    <property name="cacheLevelName"
      value="CACHE_CONSUMER"/> 2
```

2
Enables consumer cache level

```
</bean>
```

```
<bean id="txManager" 3
      class="org.springframework.jms.connection.JmsTransactionMan
      ager">
    <property name="connectionFactory"
      ref="jmsConnectionFactory"/>
</bean>
```

3
Configures Spring JmsTransactionManager

```
<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
      value="tcp://localhost:61616"/> 4
```

④

URL to the ActiveMQ message broker

```
</bean>
```

The first thing you do is turn on `transacted` for the ActiveMQ component ①, which instructs it to use transacted-acknowledge mode. Then you need to refer to the transaction manager, Spring `JmsTransactionManager` ③, which manages transactions when using JMS messaging.

JMS, JDBC, and JTA TransactionManagers

`JmsTransactionManager` is provided out of the box from the Spring JMS component and is able to handle transactions only with JMS resources. Section 12.3.3 covers using JDBC as a transaction instead of using JMS. Section 12.3.2 covers using both JMS and JDBC together, which requires using the JTA transaction manager.

The JMS transaction manager needs to know how to connect to the JMS broker, which refers to the connection factory. In the `jmsConnectionFactory` definition, you configure the `brokerURL` ④ to point at the JMS broker.

Because you enabled the transaction ①, the default behavior of the JMS component is to let the JMS consumer be in autocaching mode. The caching mode has a high impact on the performance, and it's highly recommended to configure the mode to be a cache consumer ②. This can safely be done with most JMS message brokers such as ActiveMQ. Notice that when using JTA transactions the consumer can't always be cached. The level of cache that can be in use depends on the involved message brokers, databases, and transaction managers. A rule of

thumb is to always cache the consumer for non-JTA transactions, and use auto mode for other combinations.

TIP The `cacheLevelName` option on the ActiveMQ component significantly affects performance. We recommend that you read what we just covered one more time.

So far, you've reconfigured only beans in the XML file, which is mandatory when using Spring. In Camel itself, you haven't yet configured anything in relation to transactions. Camel offers great convention over configuration for transaction support, so all you have to do is add `<transacted/>` to the route, after `<from>`, as highlighted here:

```
<camelContext id="camel"
  xmlns="http://camel.apache.org/schema/spring">
  <propertyPlaceholder id="properties"
    location="camelinaction/sql.properties"/>
  <route id="partnerToDB">
    <from uri="activemq:queue:partners"/>
    <transacted>
      <bean ref="partner" method="toMap"/>
      <to uri="sql:{${sql-insert}}?dataSource=#myDataSource"/>
    </route>
  </camelContext>
```

When you specify `<transacted/>` in a route, Camel uses transactions for that particular route and any other routes that the message may undertake. Here in the beginning you'll use transactions with only one route; in section 12.3.5, you'll go further down the road and cover transactions with multiple routes.

When a route is specified as `<transacted/>`, then under the hood Camel looks up the Spring transaction manager and uses it. This is the convention over configuration kicking in.

Using `transacted` in the Java DSL is just as easy, as shown here:

```
from("activemq:queue:partners").routeId("partnerToDB")
    .transacted()
    .bean("partner", "toMap")
    .to("sql:{sql-insert}?dataSource=#myDataSource");
```

The convention over configuration applies only when you have a single Spring transaction manager configured. In more complex scenarios, where you either use multiple transaction managers or transaction propagation policies, you have to do additional configuration.

Look at the source code to better understand how all this fits together

The book's source code contains this example in the chapter12/riderautoparts-partner directory. At this time, you may want to open the source code in your Java editor and take a look to help you better understand how the configuration in the Spring XML file fits together. In the following section, you'll learn how to test this example.

In this example, all you had to do to configure Camel was to add `<transacted/>` in the route. You relied on the transactional default configurations, which greatly reduces the effort required to set up the various bits. Section 12.3 delves deeper into configuring transactions.

Let's see if this configuration is correct by testing it.

12.2.3 TESTING TRANSACTIONS

When you test Camel routes using transactions, it's common to test with live resources, such as a real JMS broker and a database. For example, the book's source code uses Apache ActiveMQ and Derby as live resources. We picked these because they can be easily downloaded using Apache Maven and they're lightweight and embeddable, which makes them perfect for unit

testing. No up-front work is needed to install them. To demonstrate how this works, we'll return to the Rider Auto Parts example.

The last time you ran a unit test, you lost the message when there was no connection to the database. Let's try that unit test again, but this time with transactional support. You can do this by running the following Maven goal from the chapter12/riderautoparts-partner directory:

```
mvn test -Dtest=RiderAutoPartsPartnerTransactedTest
```

When you run the unit test, you'll notice a lot of stacktraces printed on the console, and they'll contain the following snippet:

```
2017-10-22 10:08:38,168 [sumer[partners]] WARN
TransactionErrorHandler Transaction rollback (0x40fcf6c1)
redelivered(false) for(MessageId: ID:davsclaus-pro-61646-
1427015316080-7:1:2:1:1 on ExchangeId:ID-davsclaus-pro-
61645-1427015315762-0-3) caught:
java.net.ConnectException: Cannot connect to the database
2017-10-22 10:08:38,173 [sumer[partners]] WARN
EndpointMessageListener Execution of JMS message listener
failed. Caused by:[org.apache.camel.RuntimeCamelException -
java.net.ConnectException: Cannot connect to the database]
```

You can tell from the stacktrace that `TransactionErrorHandler` (shown in bold) logged an exception at the `WARN` level, with a transaction rollback message. Just below the `EndpointMessageListener` (also shown in bold) is logged a `WARN` message with the caused exception and stacktrace. The `EndpointMessageListener` is a `javax.jms.MessageListener`, which is invoked when a new message arrives on the JMS destination. It'll roll back the transaction if an exception is thrown.

Where is the message now? It should be on the JMS queue, so let's add a little code to the unit test to check that.

Uncomment the following code at the end of the unit test method with the name `testNoConnectionToDatabase` (in the `RiderAutoPartsPartnerTransactedTest` class) in [listing 12.3](#):

```
Object body =
```

```
consumer.receiveBodyNoWait("activemq:queue:partners");
assertNotNull("Should not lose message", body);
```

Now you can run the unit test to ensure that the message wasn't lost—and the unit test will fail with this assertion error:

```
java.lang.AssertionError: Should not lose message
    at org.junit.Assert.fail(Assert.java:74)
    at org.junit.Assert.assertTrue(Assert.java:37)
    at org.junit.Assert.assertNotNull(Assert.java:356)
    at
camelinaction.RiderAutoPartsPartnerTransactedTest.testNoCon
nectionTo
Database(RiderAutoPartsPartnerTransactedTest.java:97)
```

You're using transactions, and they've been configured correctly, but the message is still being lost. What's wrong? If you dig into the stacktraces, you'll discover that the message is always redelivered six times, and then no further redelivery is conducted.

TIP If you're using Apache ActiveMQ, we recommend you pick up a copy of *ActiveMQ in Action*, by Bruce Snyder et al. (Manning, 2011).

What happens is that ActiveMQ performs the redelivery according to its default settings, which say it will redeliver at most six times before giving up and moving the message to a dead letter queue. This is, in fact, the Dead Letter Channel EIP. You may remember that we covered this in chapter 11 (look back to figure 11.4). ActiveMQ implements this pattern, which ensures that the broker won't be doomed by a poison message that can't be successfully processed and that would cause arriving messages to stack up on the queue.

NOTE Section 12.3.4 covers transaction redeliveries and nuances in much more depth (for example, when using ActiveMQ as the message broker).

Instead of looking for the message on the partner's queue, you should look for the message in the default ActiveMQ dead letter queue, which is named `ActiveMQ.DLQ`. If you change the code accordingly (as shown in bold), the test will pass:

```
Object body =
consumer.receiveBody("activemq:queue:ActiveMQ.DLQ", 5000);
assertNotNull("Should be in ActiveMQ DLQ", body);
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

You need to do one additional test to cover the situation where the connection to the database fails only at first, but works on subsequent calls. Here's that test.

Listing 12.5 Testing a simulated rollback on the first try and a commit on the second try

```
public void testFailFirstTime() throws Exception {
    RouteBuilder rb = new RouteBuilder() {
        public void configure() throws Exception {
            interceptSendToEndpoint("sql:*")
                .choice() ①
```

①

Causes failure first time

```
.when(header("JMSRedelivered").isEqualTo("false"))
    .throwException(new ConnectException(
        "Cannot connect to the
database"));
}
};

context.getRouteDefinition("partnerToDB")
.adviceWith(context, rb); ②
```

②

Advises the route with the simulated error route from **①**

```
NotifyBuilder notify = new NotifyBuilder(context)
    .whenDone(1 +
1).create();      ③
```

③

Sets up NotifyBuilder for the number of messages you expect

```
String sql = "select count(*) from partner_metric";
assertEquals(0, jdbc.queryForInt(sql));

String xml = "<?xml version=\"1.0\"?>
    + <partner id=\"123\">
<date>201702250815</date>
    + <code>200</code><time>4387</time>
</partner>";

template.sendBody("activemq:queue:partners", xml);

assertTrue(notify.matches(10, TimeUnit.SECONDS));      ④
```

④

Waits for routing to be done

```
assertEquals(1, jdbc.queryForInt(sql));

Object dlq =
consumer.receiveBody("activemq:queue:ActiveMQ.DLQ", 1000);
assertNull("Should not be in the DLQ", dlq);      ⑤
```

⑤

Asserts message not in DLQ

```
}
```

The idea is to throw `ConnectionException` only the first time. You do this by relying on the fact that any message consumed from a JMS destination has a set of standard JMS headers, and the `JMSRedelivered` header is a boolean type indicating whether the JMS message is being redelivered.

The interceptor logic is done in a Camel RouteBuilder, so you have the full DSL at your disposal. You use the Content-Based Router EIP ❶ to test the JMSRedelivered header and throw the exception only if it's false, which means it's the first delivery. The route must then be advised ❷ to use the route that simulates the error. The rest of the unit test should verify correct behavior, so you first check that the database is empty before sending the message to the JMS queue. Then with help from NotifyBuilder, you wait for the route to be done ❸, expecting two messages in total ❹; the first message, which should fail, and then the second redelivered message that should complete. After completion, you check that the database has one row. Because you previously were tricked by the JMS broker's dead letter queue, you also check that it's empty ❺.

Example for Apache Karaf users

The RiderAutoParts Partner example that was used in this section is also provided as an example for users of Apache Karaf, ServiceMix, or JBoss Fuse. The book's source code contains the example in the `riderautoparts-partner-karaf` directory. The source includes a `readme.md` file, which details how to install and try this example on Apache Karaf.

The preceding example uses *local transactions*, because they're based on using only a single resource in the transaction; Spring was orchestrating only the JMS broker. But there was also the database resource, which in the example wasn't under transactional control. Using both the JMS broker and the database as resources participating in the same transaction requires more work, and the next section explains about using single and multiple resources in a transaction. First, we'll look at this from the EIP perspective.

12.3 The Transactional Client EIP

The Transactional Client EIP distills the problem of how a client can control transactions when working with messaging. It's depicted in [figure 12.5](#).

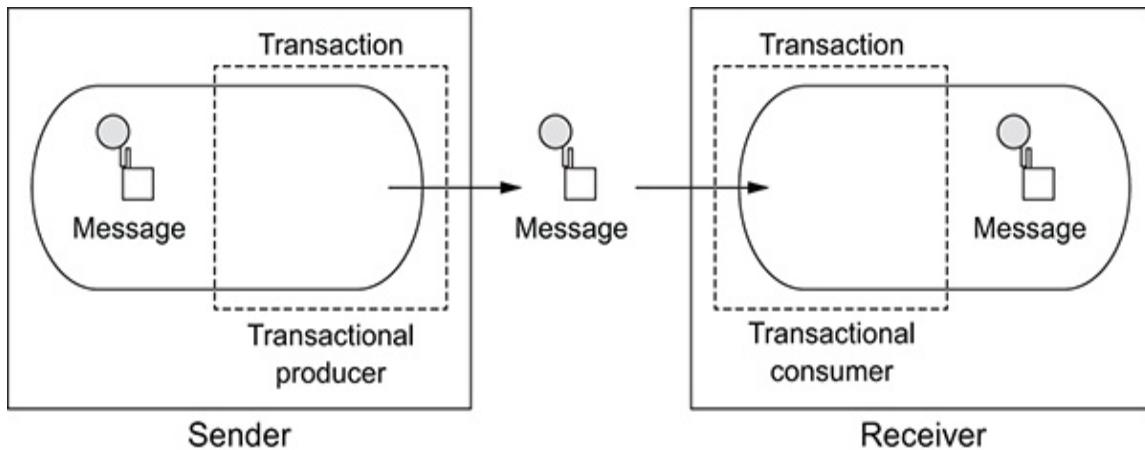


Figure 12.5 A transactional client handles the client's session with the receivers so the client can specify transaction boundaries that encompass the receiver.

[Figure 12.5](#) shows how this pattern was portrayed in Gregor Hohpe and Bobby Woolf's *Enterprise Integration Patterns* book, so it may be a bit difficult to understand how it relates to using transactions with Camel. What the figure shows is that both a sender and a receiver can be transactional by working together. When a receiver initiates the transaction, the message is neither sent nor removed from the queue until the transaction is committed. When a sender initiates the transaction, the message isn't available to the consumer until the transaction has been committed. [Figure 12.6](#) illustrates this principle.

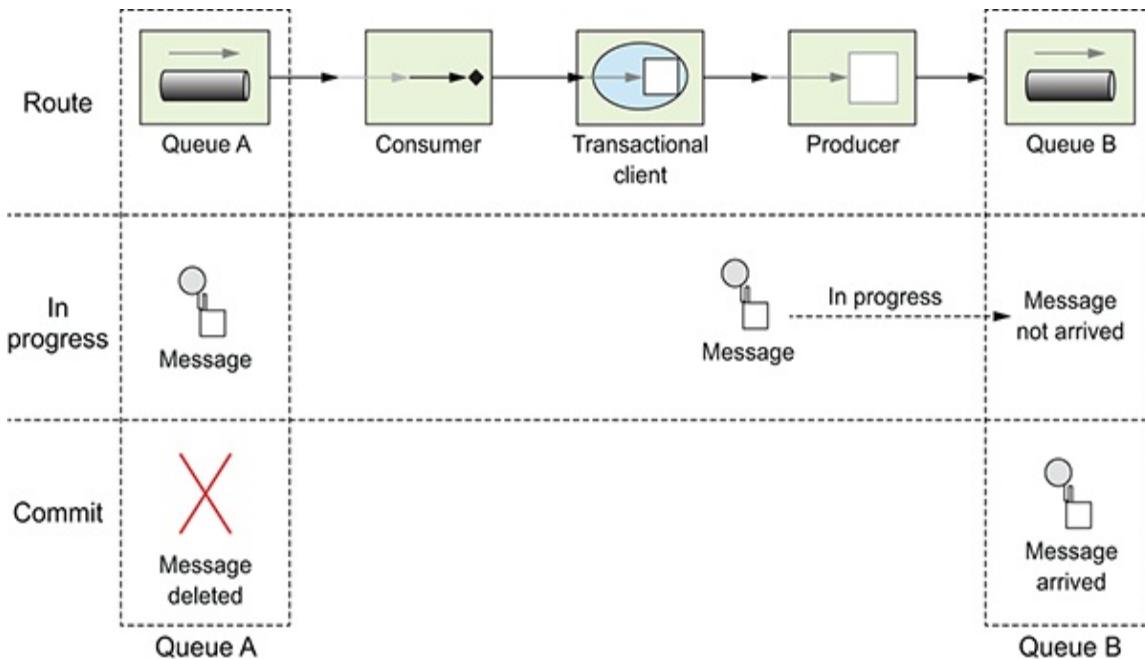


Figure 12.6 A message is being moved from queue A to queue B. Transactions ensure that the message is moved in what appears to be an atomic operation.

The top section of figure 12.6 illustrates the route using EIP icons, with a message being moved from queue A to B using a transaction. The remainder of the figure shows a use-case when one message is being moved.

The middle section shows a snapshot in time when the message is being moved. The message still resides in queue A and hasn't yet arrived in queue B. The message stays in queue A until a commit is issued, which ensures that the message isn't lost in case of a severe failure.

The bottom section shows the situation when a commit has been issued. The message is then deleted from queue A and inserted into queue B. Transactional clients make this whole process appear as an atomic, isolated, and consistent operation.

When talking about transactions, you need to distinguish between single- and multiple-resource transactions. The former are also known as *local* transactions, and the latter as *global* transactions. In the next two sections, we'll look at these two flavors.

12.3.1 USING LOCAL TRANSACTIONS

Figure 12.7 depicts the situation of using a single resource—the JMS broker. In this situation, `JmsTransactionManager` orchestrates the transaction with the single participating resource, which is the JMS broker ❶. `JmsTransactionManager` from Spring can orchestrate only JMS-based resources, so the database isn't orchestrated.

In the Rider Auto Parts example in section 12.1, the database didn't participate as a resource in the transaction, but the approach seemed to work anyway. That's because if the database decides to roll back the transaction, it will throw an exception that the Camel `TransactionErrorHandler` propagates back to `JmsTransactionManager`, which reacts accordingly and issues a rollback ❷.

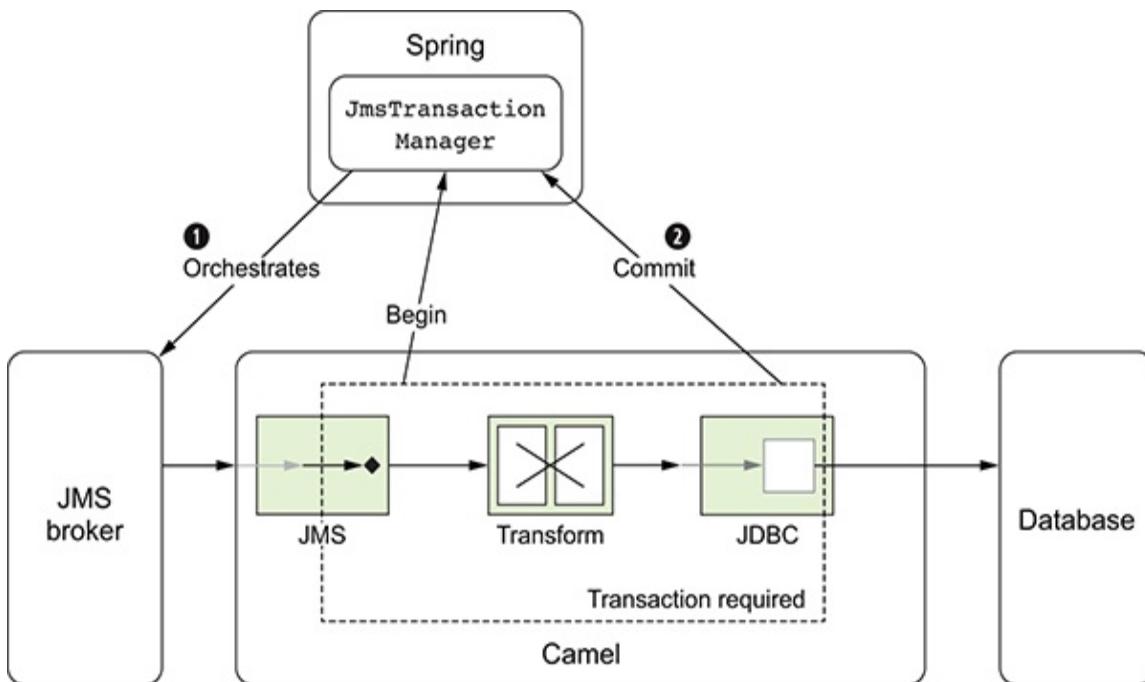


Figure 12.7 Using `JmsTransactionManager` as a single resource in a transaction.
The database isn't a participant in the transaction.

This scenario isn't exactly equivalent to enrolling the database in the transaction, because it still has failure scenarios that could leave the system in an inconsistent state. For example, the JMS broker could fail after the database is successfully updated, but

before the JMS message is committed. To be absolutely sure that both the JMS broker and the database are in sync in terms of the transaction, you must use global transactions. Let's take a look at that now.

12.3.2 USING GLOBAL TRANSACTIONS

Using transactions with a single resource is appropriate when a single resource is involved. But the situation changes when you need to span multiple resources in the same transaction, such as JMS and JDBC resources, as depicted in [figure 12.8](#).

In this figure, we've switched to using `JtaTransactionManager`, which handles multiple resources. Camel consumes a message from the queue, and a `begin` is issued **①**. The message is processed, updating the database, and it completes successfully **②**.

What is `JtaTransactionManager`, and how is it different from `JmsTransactionManager` used in the previous section ([figure 12.7](#))? To answer this, you first need to learn a bit about global transactions and where the Java Transaction API (JTA) fits in.

In Java, JTA is an implementation of the XA standard protocol, which is a global transaction protocol. To be able to use XA, the resource drivers must be XA-compliant, which some JDBC and most JMS drivers are. JTA is part of the Java EE specification, which means that any Java EE-compliant application server must provide JTA support. This is one of the benefits of Java EE servers, which have JTA out of the box, unlike some lightweight alternatives, such as Apache Tomcat.

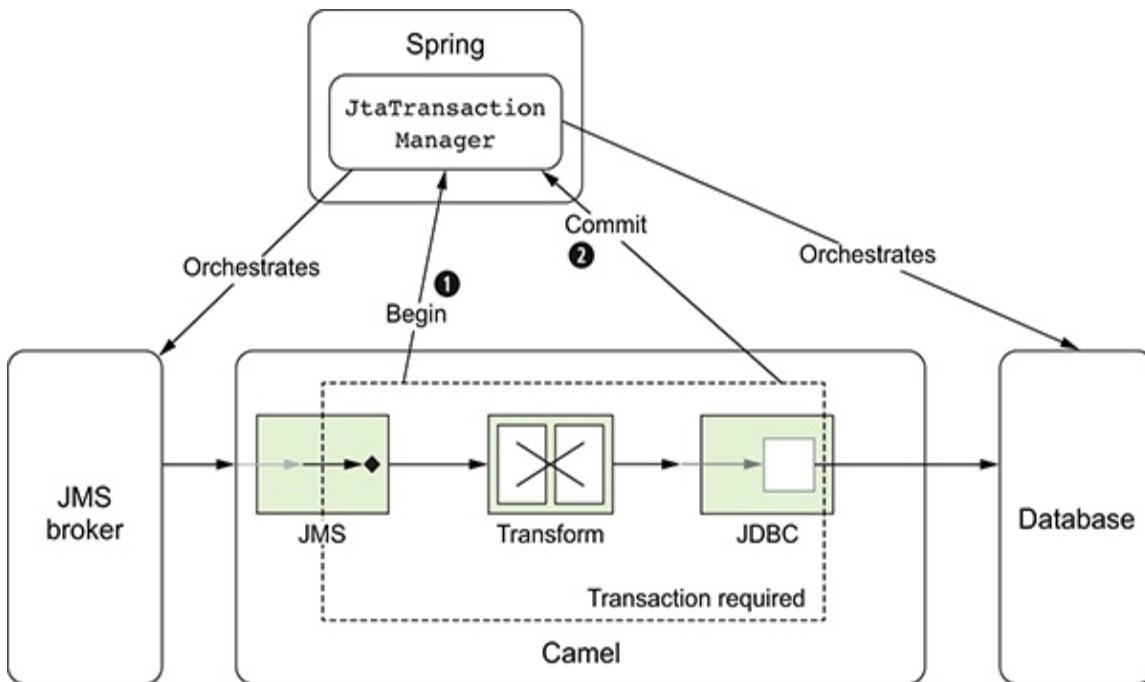


Figure 12.8 Using `JtaTransactionManager` with multiple resources in a transaction.
Both the JMS broker and the database participate in the transaction.

JTA is also available in OSGi containers such as Apache Karaf, ServiceMix, or JBoss Fuse. Using JTA outside a Java EE server takes some work to set up because you have to find and use a JTA transaction manager, such as one of these:

- *Atomikos (external third party)*—<http://www.atomikos.com>
- *Narayana (JBoss AS/WildFly)*—<http://narayana.io>
- *Apache Geronimo (Java EE)*—<http://geronimo.apache.org>
- *Apache Aries (OSGi platform)*—<http://aries.apache.org>

Then you need to figure out how to install and use it in your container and unit tests. The good news is that using JTA with Camel and Spring is just a matter of configuration.

NOTE For more information on JTA, see the Wikipedia page on the subject:
http://en.wikipedia.org/wiki/Java_Transaction_API. XA is also briefly discussed here: http://en.wikipedia.org/wiki/X/Open_XA.

When using JTA (XA), there are a couple of differences from using local transactions. First, you have to use XA-capable drivers, which means you have to use

`ActiveMQXAConnectionFactory` to let ActiveMQ participate in global transactions:

```
<bean id="jmsXaConnectionFactory"
    class="org.apache.activemq.ActiveMQXAConnectionFactory">
    <property name="brokerURL"
        value="tcp://localhost:61616"/>
</bean>
```

The same applies for the JDBC driver—you need to use an XA-capable driver. You can use Atomikos to set up a pooled XA `DataSource` using an in-memory embedded Apache Derby database:

```
<bean id="myDataSource"
    class="com.atomikos.jdbc.AtomikosDataSourceBean"
        init-method="init" destroy-method="close">
    <property name="uniqueResourceName" value="partner"/>
    <property name="xaDataSourceClassName"
        value="org.apache.derby.jdbc.EmbeddedXADataSource"/>
    <property name="minPoolSize" value="1"/>
    <property name="maxPoolSize" value="5"/>
    <property name="xaProperties">
        <props>
            <prop
                key="databaseName">memory:partner;create=true</prop>
        </props>
    </property>
</bean>
```

In a real production system, you should use the JDBC driver of the vendor of your database, such as Oracle, Postgres, MySQL, or Microsoft SQL Server, that's XA-capable.

Having configured the XA drivers, you also need to use the Spring `JtaTransactionManager`. It should refer to the real XA transaction manager, which is Atomikos in this example:

```
<bean id="jtaTransactionManager"
```

```
class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManager"
ref="atomikosTransactionManager"/>
    <property name="userTransaction"
ref="atomikosUserTransaction"/>
</bean>
```

The remainder of the configuration involves configuring Atomikos itself, which you can see in the book's source code, in the file chapter12/xa/src/test/resources/camel-spring.xml.

Suppose you want to add a step to the route shown in [figure 12.8](#). You'll process the message *after* it's been inserted into the database. This additional step will influence the outcome of the transaction, whether or not it throws an exception.

Suppose it does indeed throw an exception, as portrayed in [figure 12.9](#). In this figure, the message is being routed ① and, at the last step in the route (in the bottom-right corner with the X), it fails by throwing an exception. JtaTransactionManager handles this by issuing rollbacks ② to both the JMS broker and the database. Because this scenario uses global transactions, both the database and the JMS broker will roll back, and the final result is as if the entire transaction hadn't taken place.

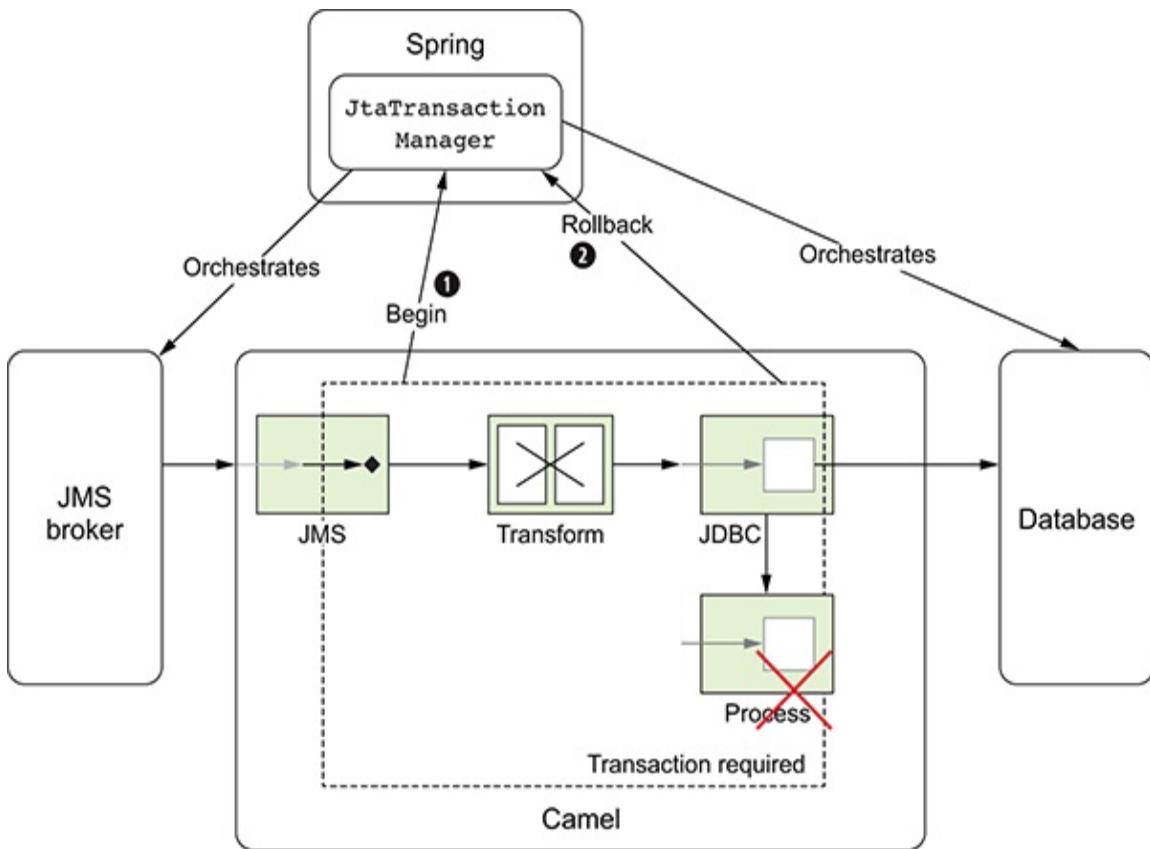


Figure 12.9 A failure to process a message at the last step in the route causes `JtaTransactionManager` to issue rollbacks to both the JMS broker and the database.

The source code for the book contains this example in the `chapter12/xa` directory. You can test it using the following Maven goals:

```
mvn test -Dtest=XACommitTest
mvn test -Dtest=XARollbackBeforeDbTest
mvn test -Dtest=XARollbackAfterDbTest
mvn test -Dtest=SpringXACommitTest
mvn test -Dtest=SpringXARollbackBeforeDbTest
mvn test -Dtest=SpringXARollbackAfterDbTest
```

Apache Karaf example with global transactions

The example is also available for Apache Karaf,

ServiceMix, or JBoss Fuse users in the chapter12/xa-karaf directory, which includes a readme.md file that details how to try the example. Because setting up global transactions with Apache Karaf is difficult, it's highly recommended to study this source code and pay attention to how transaction managers, JCA resources, and the like are defined and configured.

So far, all our examples start from a JMS resource. What if a database is the starting resource?

12.3.3 TRANSACTION STARTING FROM A DATABASE RESOURCE

Transactions can also begin from a database, and that's the topic for this section.

Now suppose you flip the JMS broker and database from the use-case in figure 12.8, which gives you figure 12.10.

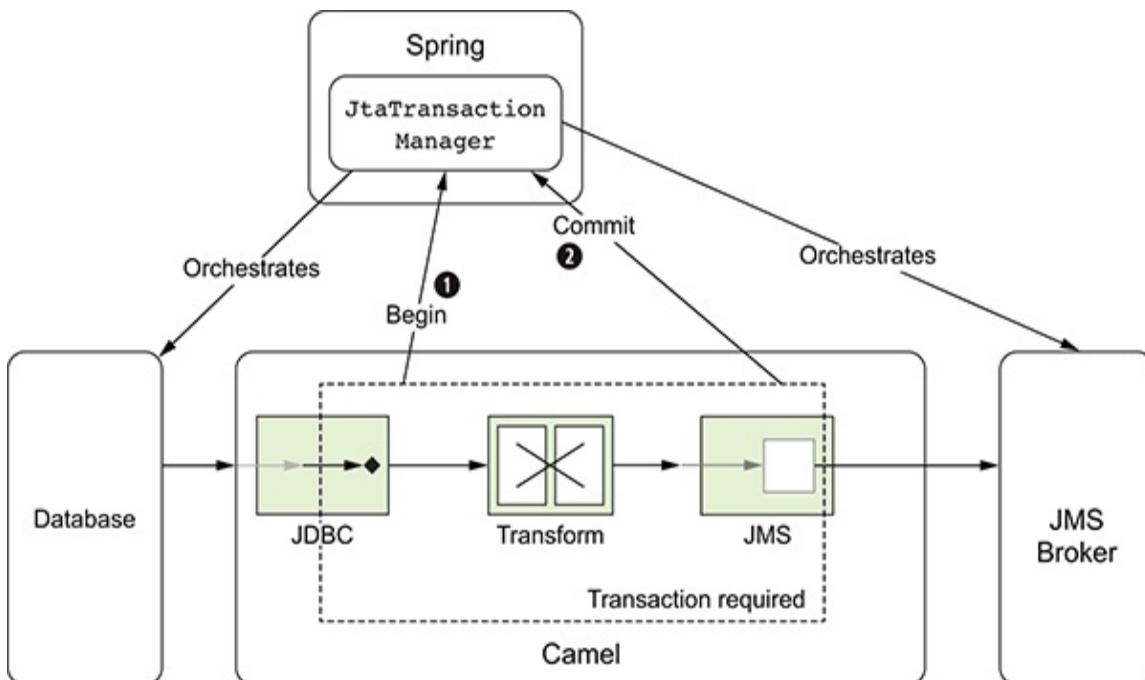


Figure 12.10 Using `JtaTransactionManager` with multiple resources in a transaction. The transaction starts from a database resource and includes the JMS broker.

Because you're using both a database and a JMS resource in the same transaction, you need to use `JtaTransactionManager` to orchestrate the transaction. The setup of the database and JMS resources in the Camel application is exactly the same as the examples from the previous section. What changes is the Camel route to start from a database.

The book's source code contains this example in the `chapter12/xa-database` directory. You can try this example by using the following goals:

```
mvn test -Dtest=SpringXACommitTest  
mvn test -Dtest=SpringXARollbackBeforeActiveMQTest  
mvn test -Dtest=SpringXARollbackAfterActiveMQTest
```

The example uses the SQL component to poll the database for new data:

```
<route>  
    <from uri="sql:{sql-from}?consumer.onConsume={{sql-delete}}>  
  
    & dataSource=#myDataSource& transacted=true"/>  
    <transacted/>  
    <log message="*** transacted ***"/>  
    <to uri="log:row"/>  
    <to uri="activemq:queue:order"/>  
</route>
```

Notice that the SQL endpoint is configured with `transacted=true`, which tells Camel that the consumer runs in transacted mode. This must be done because the SQL consumer is now the input of the route, and the consumer must be aware of the transaction being used as well. This is similar to when using JMS, where you configured transacted-acknowledge mode, as shown in [listing 12.4](#).

ONLY A DATABASE

If you take out the JMS broker from the example so you're using only the database as a single resource, you don't need to use XA. And instead of using `JtaTransactionManager`, you can use

`DataSourceTransactionManager`, which is intended for a single-resource database. The book's source code contains such an example in the `chapter12/tx-database` directory, and you can try the example by using the following goals:

```
mvn test -Dtest=SpringCommitTest  
mvn test -Dtest=SpringRollbackTest
```

NOTE We recommend that WildFly users using the resources that ship with the WildFly server look at the `wildfly-camel` documentation explaining how to use transactions with WildFly and Apache Camel: http://wildfly-extras.github.io/wildfly-camel/#_jms.

When a message is rolled back, the ACID principle of the transaction means that the outcome is as if the message didn't happen; it's all or nothing. What happens next is covered in the next section.

12.3.4 TRANSACTION REDELIVERIES

When a transaction rolls back, what happens next depends on whether the message came from a message broker or a database. The former will often have various redelivery settings you can configure, which we cover in this section. But neither a database nor the JDBC components from Camel provide transactional redelivery support. At the end of this section, we'll talk about what you can do instead.

On a message broker such as Apache ActiveMQ, message redelivery can be handled and configured at two levels:

- Consumer level (used by default)
- Broker level

By default, redelivery is controlled by the consumer: it's the duty of the consumer to keep track of the number of times a message has been redelivered, the amount of time the consumer should

delay between redelivery attempts, and whether a message is exhausted and should be moved to the ActiveMQ dead letter queue.

ActiveMQ redelivery vs. Camel redelivery

This may feel familiar, and, yes, it's also similar to what Camel's error handler is capable of doing. Many of the options you use to configure redelivery with ActiveMQ are similar to options you'd use with Camel. This isn't a surprise, as both ActiveMQ and Camel were created originally by the same group of people.

To allow the consumer to keep track of redelivery of the message, the consumer must be reused by the Camel route, and to ensure that you must configure the JMS or ActiveMQ component in Camel to cache the consumer, as shown here in bold:

```
<bean id="activemq"
    class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="transacted" value="true"/>
    <property name="transactionManager"
        ref="jtaTransactionManager"/>
    <b><property name="cacheLevelName"
        value="CACHE_CONSUMER"/></b>
</bean>
```

This is important to remember, because the JMS or ActiveMQ component will by default be conservative in transacted mode, and not cache the consumer. You may not notice a difference because when a transaction rollback happens, the message is still being redelivered. But because the consumer wasn't cached, a new consumer is created for each message received, and that consumer has no previous state about the redelivery, and as a

result there are no delays between any redeliveries. Because there are no delays between redeliveries, the redelivered messages will happen rapidly in sequence. The ActiveMQ also keeps track of the number of redelivery attempts, and when the message has failed too many times in a row, it's automatically moved to the ActiveMQ dead letter queue.

CONFIGURING CONSUMER-LEVEL REDELIVERY WITH ACTIVEMQ

But if the consumer is cached, then by default the consumer will delay each redelivery by 1 second and therefore redeliveries will be a bit slower. Because 1 second may still be too fast, you can configure a slower setting in the `brokerURL` option, as shown here:

```
<bean id="jmsXaConnectionFactory"
  class="org.apache.activemq.ActiveMQXAConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616
& jms.redeliveryPolicy.maximumRedeliveries=10
& jms.redeliveryPolicy.redeliveryDelay=5000"/>
</bean>
```

Each redelivery option must be prefixed with `jms.redeliveryPolicy`. In the preceding example, we configured maximum redeliveries to 10, and used 5 seconds as delay.

TIP ActiveMQ has other ways to configure redelivery, which you can find more about on the website:
<http://activemq.apache.org/redelivery-policy.html>.

Consumer-level redelivery conducts retries so that message ordering is maintained while messages appear as inflight on the broker. Redelivery of messages from the given queue is limited to a single consumer, as the broker is unaware that the consumer is performing redeliveries. If message ordering isn't important,

and/or higher throughput and load distribution among concurrent consumers is desired, then consumer level redelivery isn't sufficient, and broker level should be used instead.

CONFIGURING BROKER-LEVEL REDELIVERY WITH ACTIVEMQ

Broker-level redelivery is configured as an ActiveMQ plugin that's defined in the broker XML configuration file:

```
<broker xmlns="http://activemq.apache.org/schema/core"
        schedulerSupport="true">
    <plugins>
        <redeliveryPlugin fallbackToDeadLetter="true"
sendToDlqIfMaxRetriesExceeded="true">
            <redeliveryPolicyMap>
                <redeliveryPolicyEntries>
                    <defaultEntry>
                        <redeliveryPolicy maximumRedeliveries="10"
initialRedeliveryDelay="5000"                                redeliveryDelay="10000"/>
                        </defaultEntry>
                    </redeliveryPolicyEntries>
                </redeliveryPolicyMap>
            <redeliveryPlugin>
        </plugins>
```

In this example, we perform at most 10 redeliveries with 10 seconds delay in between, but the first delay is only 5 seconds. You can find more details about ActiveMQ redelivery at the ActiveMQ website.

Let's now talk about redeliveries when you start from a database.

TRANSACTION REDELIVERIES STARTING FROM DATABASE

As opposed to a message broker, a database doesn't have the concept of redelivery or dead letter queue. Therefore, you have limited options of what you can do to address repeated transaction rollbacks when starting from a database.

The book's source code contains an example that starts a

transaction from a database in the chapter12/tx-database directory. There's a rollback example that you can try using the following Maven goal:

```
mvn test -Dtest=SpringRollbackTest
```

The SQL consumer will, by default, poll the database every half second. Because the example is designed to fail, it'll throw an exception during processing that causes a transactional rollback. On the next poll, the same thing happens again, and again, and so on. The repeated number of consecutive rollbacks may stress the database and your monitoring system, which may detect a failure every half second. In those situations, you can configure the SQL consumer to back off polling so frequently, as shown here:

```
<from uri="sql:{{sql-from}}?consumer.onConsume={{sql-
delete}}>

& dataSource=#myDataSource& transacted=true

& backoffMultiplier=5& backoffErrorThreshold=1"/>
```

The options `backoffErrorThreshold=1` and `backoffMultiplier=5` tell Camel to multiply the polling interval by 5 after there's been one error, so the polling will happen every $0.5 \times 5 = 2.5$ seconds. You want to use this when the error may be recoverable, and maybe after a while the message can be processed successfully, and the transaction can commit. When this happens, the back-off will return to normal, and poll every half second again.

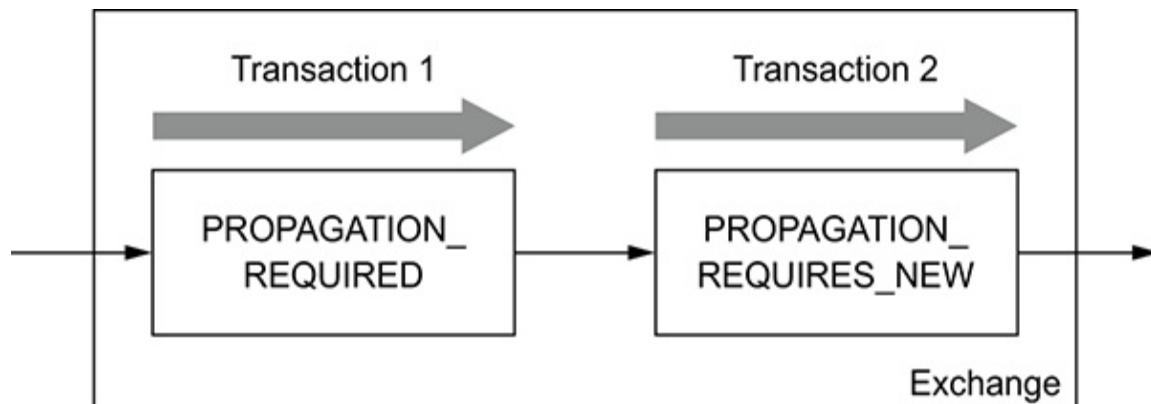
TIP The SQL endpoint also provides a back-off capability when there's no data, using the `backoffIdleThreshold` option. For example, `backoffIdleThreshold=3` will trigger back-off after three consecutive polls with no data.

So far, we've used convention over configuration when configuring transactions in Camel, by just adding `<transacted/>`

to the route. This is often all you'll need to use, but at times you'll need more fine-grained control, such as using two different transaction propagations.

12.3.5 USING DIFFERENT TRANSACTION PROPAGATIONS

In some situations, you may need to use multiple transactions with the same exchange, as illustrated in [figure 12.11](#).



[Figure 12.11](#) Using two independent transactions in a single exchange

In [figure 12.11](#), an exchange is being routed in Camel. It starts off using the required transaction, and then you need to use another transaction that's independent of the existing transaction. You can do this by using `PROPAGATION_REQUIRES_NEW`, which will start a new transaction regardless of whether an existing transaction exists. When the exchange completes, the transaction manager will issue commits or rollbacks to these two transactions, which ensures that they both complete at the same time. Because two transaction legs are in play, they can have different outcomes; for example, transaction 1 can roll back, while transaction 2 commits, and vice versa.

In Camel, a route can be configured to use only at most one transaction propagation, which means [figure 12.11](#) must use two routes. The first route uses `PROPAGATION_REQUIRED`, and the second route uses `PROPAGATION_REQUIRES_NEW`.

Suppose you have an application that updates orders in a database. The application must store all incoming orders in an

audit log and then it either updates or inserts the order in the order database. The audit log should always insert a record, even if subsequent processing of the order fails. Implementing this in Camel should be done using two routes, as shown in the following listing.

Listing 12.6 Using two independent transactions in a single exchange

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
  <route>  
    <from uri="activemq:queue:inbox"/>  
    <transacted ref="required"/> 1
```

1

Requires transaction

```
      <to uri="direct:audit"/> 2
```

2

Calls the audit-log route

```
      <to uri="direct:order"/> 3
```

3

Calls the insert-order route

```
    <to uri="activemq:queue:order"/>  
  </route>  
  
  <route>  
    <from uri="direct:audit"/>  
    <transacted ref="requiresNew"/> 4
```

4

Uses a new transaction for the audit-log route

```
    <bean ref="auditLogService" method="insertAuditLog"/>
```

```
</route>

<route>
    <from uri="direct:order"/>
    <transacted ref="mandatory"/> 5
```

5

Uses the existing parent transaction for the insert-order route

```
<bean ref="orderService" method="insertOrder"/>
</route>
</camelContext>
```

The first route uses PROPAGATION_REQUIRED to start transaction 1 **1**. Then the audit-log route is called **2**, which uses PROPAGATION_REQUIRES_NEW **4** to start transaction 2 that runs independently from transaction 1. After the message has been inserted into the audit log, the insert-order route is called **3**. This route will reuse the existing transaction 1 from the parent route **5**, which is specified by using the mandatory propagation property.

In listing 12.6, we're using three propagation behaviors, which are configured as shown in the following listing.

Listing 12.7 Configure three propagation behavior beans in XML DSL

```
<bean id="required"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager"
      ref="jtaTransactionManager"/>
    <property name="propagationBehaviorName"
      value="PROPAGATION_REQUIRED"/>
  </bean>

  <bean id="requiresNew"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager"
```

```

    ref="jtaTransactionManager"/>
    <property name="propagationBehaviorName"
              value="PROPAGATION_REQUIRES_NEW"/>
</bean>

<bean id="mandatory"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager"
              ref="jtaTransactionManager"/>
    <property name="propagationBehaviorName"
              value="PROPAGATION_MANDATORY"/>
</bean>
```

The book's source code contains this example in the chapter12/propagation directory, which you can try using the following Maven goal:

```

mvn test -Dtest=PropagationTest
mvn test -Dtest=PropagationRollbackLastTest
mvn test -Dtest=SpringPropagationTest
mvn test -Dtest=SpringPropagationRollbackLastTest
```

Four unit tests demonstrate different scenarios, as described in table [12.1](#).

Table 12.1 Overview of what happens when transaction 1 or 2 either commits or rolls back

Test method	Tran sa cti on 1	Tran sa cti on 2	Comment
testWithCamel	C o m m it	C o m m it	The message is successfully processed. In the database, one row is inserted in both the order and audit-log tables.
testWithDockey	R ol lba c	C o m m it	The message fails during order validation and is rolled back. In the database, no row is inserted in the order table. But six rows are inserted in the audit-log table because transaction 2 commits. This is intended, as the use-case was designed to insert all orders into audit logs even if transaction 1 fails.

	k		
tes tAu dit Log Fail	R ol lb a c k	R ol lb a c k	The message fails during audit logging and is rolled back. In the database, no row is inserted, as both transactions roll back.
tes tAu dit Log- Ro llb ack Las t	C o m m it	R ol lb a c k	The message fails during audit logging, and only the second transaction is explicitly marked to roll back, leaving the first transaction to commit. Only the order is inserted into the database, whereas the audit log fails and is rolled back.

Keep it simple

Although you can use multiple propagation behaviors with multiple routes in Camel, do so with care. Try to design your solutions with as few propagations as possible, because complexity increases dramatically when you introduce new propagation behaviors. For example, table 12.1 lists four outcomes of using two³ propagations; if you have three propagations, there are $2^3 = 8$ combinations.

In the next section, we'll return to Rider Auto Parts and look at an example that covers a common use-case: using web services together with transactions. How do you return a custom web service response if a transaction fails?

12.3.6 RETURNING A CUSTOM RESPONSE WHEN A TRANSACTION FAILS

Rider Auto Parts has a Camel application that exposes a web service to a selected number of business partners. The partners use this application to submit orders. Figure 12.12 illustrates the

application.

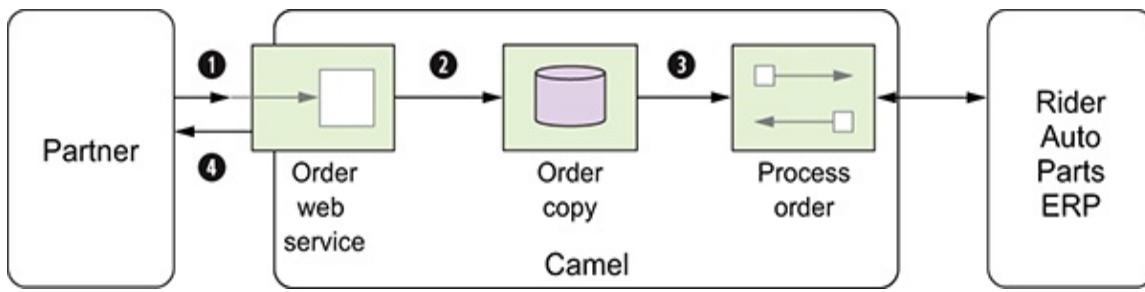


Figure 12.12 A web service used by business partners to submit orders. A copy of the order is stored in a database before it's processed by the ERP system.

As you can see, the business partners invoke a web service to submit an order ①. The received order is stored in a database for audit purposes ②. The order is then processed by the enterprise resource planning (ERP) system ③, and a reply is returned to the waiting business partner ④.

The web service is deliberately kept simple so partners can easily use it with their IT systems. A single return code indicates whether the order succeeded or failed. The following code snippet is part of the WSDL definition for the reply (`outputOrder`):

```
<xs:element name="outputOrder">
    <xs:complexType>
        <xs:sequence>
            <xs:element type="xs:string" name="code"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

The `code` field should contain `ok` if the order was accepted; any other value is considered a failure. The Camel application must deal with any thrown exceptions and return a custom failure message, instead of propagating the thrown exception back to the web service.

Your Camel application needs to do the following three things:

- Catch the exception and handle it to prevent it from propagating back

- Mark the transaction to roll back
- Construct a reply message with a code value of ERROR

Camel can support such complex use-cases because you can use `onException`, which you learned about in chapter 11.

CATCHING AND HANDLING THE EXCEPTION

What you do is add an `onException` to the Camel-Context, as shown here:

```
<onException>
    <exception>java.lang.Exception</exception>
    <handled><constant>true</constant></handled>
    <transform><method bean="order" method="replyError"/>
</transform>
    <rollback markRollbackOnly="true"/>
</onException>
```

You first tell Camel that this `onException` should trigger for any kind of exception that's thrown. You then mark the exception as handled, which removes the exception from the exchange, because you want to use a custom reply message instead of the thrown exception.

ROLLING BACK A TRANSACTION

To roll back the transaction, you mark it for rollback using a single line of code:

```
<rollback markRollbackOnly="true"/>
```

By doing this, you trigger a rollback without an exception being thrown using the `markRollbackOnly` attribute. The `<rollback/>` definition must always be at the end of `onException` because it stops the message from being further routed. That means you *must* have prepared the reply message before you issue the `<rollback/>`.

Rollback strategies

You can roll back a transaction in several ways. By default, if any unhandled exception happens during routing, the transaction error handler detects it and stops routing, and the exception is propagated back to the transaction manager that marks the transaction for rollback. Unhandled exceptions trigger a transactional rollback. In the Camel routes, you can make rollbacks obvious by using `rollback` in the DSL. The `rollback` is merely a facade for throwing an exception of the type

`org.apache.camel.RollbackExchangeException`. To do this in XML DSL, you can specify the rollback with an optional message:

```
<rollback message="Forced being rolled back"/>
```

And in Java DSL, the same statement is as follows:

```
rollback("Forced being rolled back")
```

CONSTRUCTING THE REPLY MESSAGE

To construct the reply message, you use the `order` bean, invoking its `replyError` method:

```
public OutputOrder replyError(Exception cause) {  
    OutputOrder error = new OutputOrder();  
    error.setCode("ERROR: " + cause.getMessage());  
    return error;  
}
```

This is easy to do, as you can see. You first define the `replyError` method to have an `Exception` as a parameter—this will contain the thrown exception. You then create the `outputorder` object, which you populate with the `ERROR` text and the exception message.

The book's source code contains this example in the chapter12/riderautoparts-order directory. You can start the application using the following Maven goal:

```
mvn camel:run
```

Then you can send web service requests to <http://localhost:9000/order>. The WSDL is accessible at <http://localhost:9000/order?wsdl>.

To work with this example, you need to use web services. SoapUI (www.soapui.org) is a popular application for testing with web services. It's also easy to set up and get started. You create a new project and import the WSDL file from <http://localhost:9000/order?wsdl>. Then you create a sample web service request and fill in the request parameters, as shown in figure 12.13. You then send the request by clicking the green Play button, and it will display the reply in the pane on the right side.

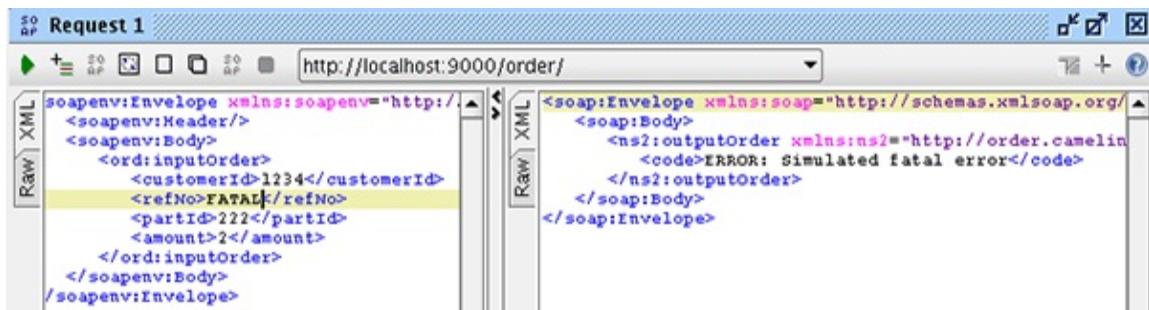


Figure 12.13 A web service message causes the transaction to roll back, and a custom reply message is returned.

In the example in figure 12.13, we caused a failure to occur. Our example behaves according to what you specify in the refNo field. You can force different behavior by specifying either FATAL or FAIL-ONCE in the refNo field. Entering any other value will cause the request to succeed. As figure 12.13 shows, we entered FATAL, which causes an exception to occur and an ERROR reply to be returned.

So far, we've been using resources that support transactions, such as JMS and JDBC, but the majority of components don't

support transactions. What can you do instead? The next section looks at compensating when transactions aren't supported.

12.4 Compensating for unsupported transactions

The number of resources that can participate in transactions is limited; they're mostly confined to JMS- and JDBC-based resources. This section covers what you can do to compensate for the absence of transactional support in other resources.

Compensation, in Camel, involves the unit-of-work concept.

First, you'll look at how a unit of work is represented in Camel and how you can use this concept. Then we'll walk through an example demonstrating how the unit of work can help simulate the orchestration of a transaction manager. We'll also discuss how you can use a unit of work to compensate for the lack of transactions by doing the work that a transaction manager's rollback would do in the case of failure.

12.4.1 INTRODUCING UNITOFWORK

Conceptually, a *unit of work* is a batch of tasks as a single coherent unit. The idea is to use the unit of work as a way of mimicking transactional boundaries.

In Camel, a unit of work is represented by the `org.apache.camel.spi.UnitOfWork` interface, offering a range of methods, including the following:

```
void addSynchronization(Synchronization synchronization);
void removeSynchronization(Synchronization
synchronization);
void done(Exchange exchange);
```

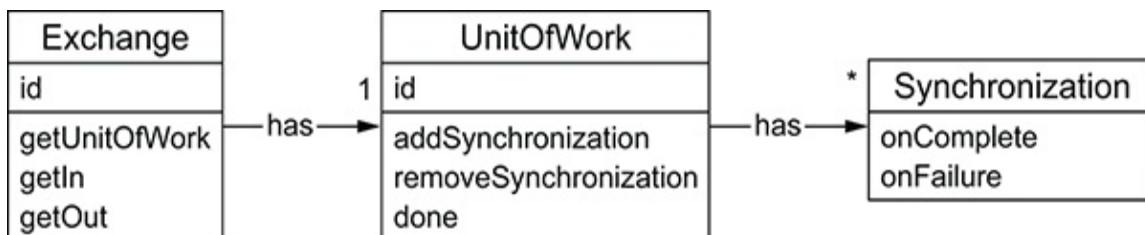
The `addSynchronization` and `removeSynchronization` methods are used to register and unregister a `Synchronization` (a callback), which we'll look at shortly. The `done` method is invoked when the unit of work is complete, and it invokes the registered callbacks.

The Synchronization callback is the interesting part for Camel end users because it's the interface you use to execute custom logic when an exchange is complete. It's represented by the `org.apache.camel.spi.Synchronization` interface and offers these two methods:

```
void onComplete(Exchange exchange);  
void onFailure(Exchange exchange);
```

When the exchange is done, either the `onComplete` or `onFailure` method is invoked, depending on whether the exchange failed.

[Figure 12.14](#) illustrates how these concepts are related to each other. As you can see, each Exchange has exactly one `UnitOfWork`, which you can access using the `getUnitOfWork` method from the Exchange. The `UnitOfWork` is private to the Exchange and isn't shared with others.



[Figure 12.14](#) An Exchange has one `UnitOfWork`, which in turn has from zero to many `Synchronizations`.

How `UnitOfWork` is orchestrated

Camel will automatically inject a new `UnitOfWork` into an Exchange when it's routed. This is done by an internal processor, `UnitOfWorkProcessor`, which is involved in the start of every route. When the Exchange is done, this processor invokes the registered synchronization callbacks. The `UnitOfWork` boundaries are always at the beginning and end of the Exchange being routed.

When an Exchange is done being routed, you hit the end

boundary of the `UnitOfWork`, and the registered synchronization callbacks are invoked one by one. This is the same mechanism the Camel components use to add their custom synchronization callbacks to the `Exchange`. For example, the File and FTP components use this mechanism to perform *after-processing* operations such as moving or deleting processed files.

TIP Use Synchronization callbacks to execute any after-processing you want done when the `Exchange` is complete.

A good way of understanding how this works is to review an example, which we'll do now.

12.4.2 USING SYNCHRONIZATION CALLBACKS

Rider Auto Parts has a Camel application that sends email messages containing invoice details to customers. First, the email content is generated, and then, before the email is sent, a backup of the email is stored in a file for reference. Whenever an invoice is to be sent to a customer, the Camel application is involved. [Figure 12.15](#) illustrates the application.

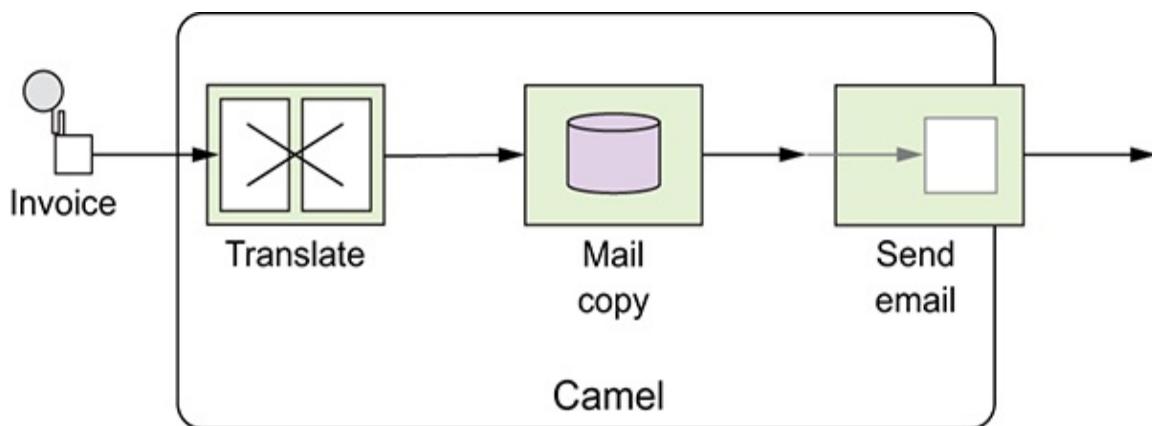


Figure 12.15 Emails are sent to customers listing their invoice details. Before the email is sent, a backup is stored in the filesystem.

Imagine what would happen if there were a problem sending an email. You can't use transactions to roll back, because filesystem resources can't participate in transactions. Instead, you can

perform custom logic, which *compensates* for the failure by deleting the file.

The compensation logic is trivial to implement, as shown here:

```
public class FileRollback implements Synchronization {  
  
    public void onComplete(Exchange exchange) {}  
  
    public void onFailure(Exchange exchange) {  
        String name = exchange.getIn().getHeader(  
            Exchange.FILE_NAME_PRODUCED,  
        String.class);  
        LOG.warn("Failure occurred so deleting backup file:  
" + name);  
        FileUtils.deleteFile(new File(name));  
    }  
}
```

In the `onFailure` method, you delete the backup file, retrieving the filename used by the Camel File component from the `Exchange.FILE_NAME_PRODUCED` header.

TIP Camel offers the `org.apache.camel.support.SynchronizationAdapter` class, which is an empty `Synchronization` implementation having the need for end users to override only the `onComplete` or `onFailure` method as needed.

What you must do next is instruct Camel to use the `FileRollback` class to perform this compensation. To do so, you can add it to the `UnitOfWork` by using the `addSynchronization` method, which was depicted in [figure 12.15](#). This can be done using the Java DSL as highlighted in the following listing.

[Listing 12.8](#) Using `UnitOfWork` as a transactional rollback from inlined Processor

```
public void configure() throws Exception {
```

```
from("direct:confirm")
    .process(new Processor() { ①
```

①

Inlined Processor

```
    public void process(Exchange exchange) throws
Exception { ①
    exchange.getUnitOfWork() ①
        .addSynchronization(new FileRollback()); ①
    } ①
}
.bean(OrderService.class, "createMail")
.log("Saving mail backup file")
.to("file:target/mail/backup")
.log("Trying to send mail to ${header.to}")
.bean(OrderService.class, "sendMail")
.log("Mail sent to ${header.to}");
}
```

The book's source code contains this example in the chapter12/uow directory. You can try it using the following Maven goal:

```
mvn test -Dtest=FileRollbackTest
```

If you run the example, it will output something like the following to the console:

```
INFO route1 - Saving mail backup file
INFO route1 - Trying to send to FATAL
ERROR DefaultErrorHandler - Failed delivery for (MessageId:
ID-davsclaus-pro-local-1512914413468-1-2 ...
WARN FileRollback - Failure occurred so deleting backup
file: target/mail/backup/ID-davsclaus-pro-local-
1512914413468-1-2
```

One thing that may bother you is that you must use an inlined Processor ① to add the FileRollback class as a Synchronization. Camel offers a convenient method on the Exchange, so you could do it with less code:

```
exchange.addOnCompletion(new FileRollback());
```

And with Java 8, you can inline the Processor ❶ using lambda style, which reduces the code from six lines to only one line:

```
.process(e -> e.addOnCompletion(new FileRollback()))
```

But it still requires the inlined Processor ❶ with or without Java 8 lambda style. Isn't there a more convenient way? Yes, and that's where `onCompletion` comes into the picture.

12.4.3 USING ONCOMPLETION

`onCompletion` takes the synchronization into the world of routing, enabling you to easily add the `FileRollback` class as `Synchronization`. Let's see how it's done.

`onCompletion` is available in both Java DSL and XML DSL variations. The following listing shows how `onCompletion` is used with XML DSL.

Listing 12.9 Using `onCompletion` as a transactional rollback

```
<bean id="orderService"
      class="camelaction.OrderService"/>
<bean id="fileRollback"
      class="camelaction.FileRollback"/>

<camelContext
  xmlns="http://camel.apache.org/schema/spring">

    <onCompletion onFailureOnly="true">
```

Context scoped `onCompletion` that triggers on failure only

```
        <bean ref="fileRollback" method="onFailure"/>
    </onCompletion>

    <route>
        <from uri="direct:confirm"/>
        <bean ref="orderService" method="createMail"/>
        <log message="Saving mail backup file"/>
        <to uri="file:target/mail/backup"/>
```

```
<log message="Trying to send mail to  
${header.to}" />  
    <bean ref="orderService" method="sendMail"/>  
        <log message="Mail sent to ${header.to}" />  
    </route>  
</camelContext>
```

As you can see, `<onCompletion>` is defined as a separate Camel route. `onCompletion` will be executed right after the message is done being routed. In this example, that means after the last step in the route, the message reaches the end, and `onCompletion` is executed:

```
<log message="Mail sent to ${header.to}" />
```

Under the hood

When you add `<onCompletion>` to `<camelContext>` or `<route>`, `<onCompletion>` will be added as a subroute that's being routed from a synchronization callback. The routing engine is responsible for orchestrating this by detecting whether `<onCompletion>` is available, and if so by adding the necessary synchronization callback to the `UnitOfWork` of the exchange. This happens when the exchange is about to be routed to the original route. When the exchange is done being routed, `UnitOfWork` kicks in and calls each synchronization callback, and hence among those the callback that triggers the `<onCompletion>` subroute.

In the present situation, you're interested in executing `onCompletion` only when the exchange fails, so you can specify this by setting the `onFailureOnly` attribute to `true`.

The book's source code contains this example in the `chapter12/uow` directory; you can run the example using the following Maven goal:

```
mvn test -Dtest=FileRollbackOnCompletionTest  
mvn test -Dtest=SpringFileRollbackOnCompletionTest
```

When you run it, you'll find that it acts just like the previous example. It will delete the backup file if the exchange fails.

`onCompletion` can also be used when the exchange didn't fail. Suppose you want to log activity about exchanges being processed. For example, in the Java DSL, you could do it as follows:

```
onCompletion().bean("logService", "logExchange");
```

`onCompletion` also supports scoping, exactly the same as `onException` does at either context or route scope (as you saw in chapter 11). You could create a Java DSL-based version of [listing 12.9](#) using route-scoped `onCompletion` as follows:

```
from("direct:confirm")  
    .onCompletion().onFailureOnly()  
        .bean(FileRollback.class, "onFailure")  
    .end()  
    .bean(OrderService.class, "createMail")  
    .log("Saving mail backup file")  
    .to("file:target/mail/backup")  
    .log("Trying to send mail to ${header.to}")  
    .bean(OrderService.class, "sendMail")  
    .log("Mail send to ${header.to}");
```

You can also attach a predicate to `onCompletion` to trigger only when the predicate matches.

USING PREDICATE WITH ONCOMPLETION

Suppose you want `onCompletion` to trigger only if a file was saved. To do this, you can use `onwhen` as the predicate to check for the presence of the `Exchange.FILE_NAME_PRODUCED` header, which the file producer adds as a message header with the name of the file that was saved:

```
from("direct:confirm")  
    .onCompletion()
```

```

    .onFailureOnly().onWhen(header(Exchange.FILE_NAME_PRODUCED)
)
    .bean(FileRollback.class, "onFailure")
.end()

```

And the corresponding code example in XML DSL:

```

<onCompletion onFailureOnly="true">
    <onWhen><header>CamelFileNameProduced</header></onWhen>
    <bean ref="fileRollback" method="onFailure"/>
</onCompletion>

```

Notice you have to use the value of the Exchange.FILE_NAME_PRODUCED key (CamelFileNameProduced) in XML DSL as the header name.

onCompletion runs by default after the consumer is completed. If you want to run onCompletion before the consumer returns a response to the caller, you have to switch modes.

USING BEFORECONSUMER MODE

As part of your work for Rider Auto Parts, you recently began developing a simple API for mobile users to access information about their orders. The API exposes a REST service that can return information about an order. [Figure 12.16](#) illustrates this principle.

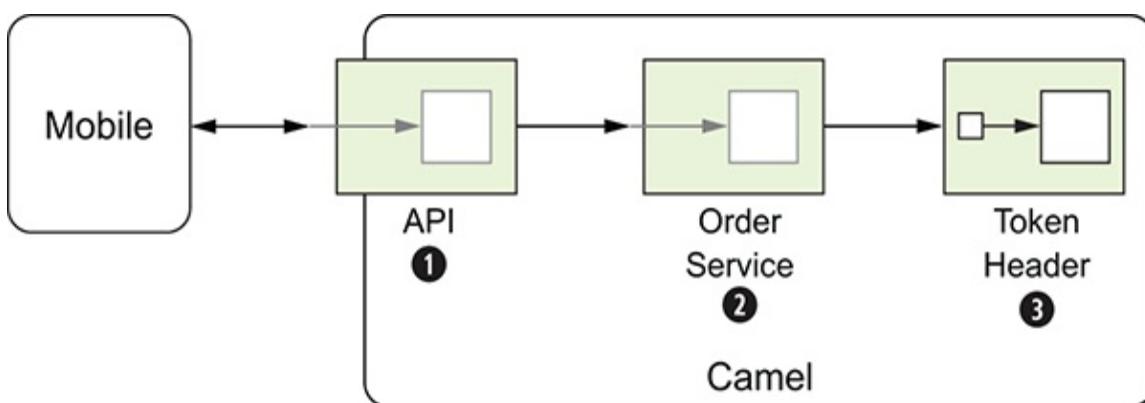


Figure 12.16 A mobile user accesses a Camel API that fetches order information and enriches the response with a token header.

Figure 12.16 shows how mobile users can access the REST service using the API ①, which then queries the ② order service

to obtain order details. If the request and order are valid, the response must be enriched with a token ❸ that the mobile client must use for subsequent queries to the API.

The API is implemented in Camel via a `RouteBuilder` class using `rest-dsl`, as shown in the following listing.

Listing 12.10 API implemented using `rest-dsl` in Camel

```
public void configure() throws Exception {  
    restConfiguration().component("netty4-http") ❶  
  
    .bindingMode(RestBindingMode.json)  
    .host("localhost").port(8080).contextPath("service");  
    onCompletion().modeBeforeConsumer() ❸  
  
    .setHeader("Token").method(TokenService.class);  
    rest()  
        .get("/order/{id}") ❷  
  
        .to("bean:orderService?method=getOrder");
```

First, you set up the rest configuration to use the `netty4-http` component as the HTTP server ❶. You're using Netty because it's a popular, fast, and lightweight networking framework; which works well in these times of microservices. The REST services are exposed on `localhost:8080/service` and use JSON

binding mode.

In Rest DSL, you define a REST service to get the order status by ID **②**, which will map to clients using the URL `http://localhost:8080/service/order/{id}`.

As part of the API, it's now a requirement that a token header is returned with a unique value. The mobile service is then required to use the token header for subsequent access to the API when other services are requested. To add the unique token header, you use `onCompletion` **③**. Switching to `BeforeConsumer` mode ensures that the header is added to the message before the REST service returns the response to the mobile client.

The book's source code contains this example in `chapter12/uow` as a Java and Spring example. The following Maven goals run the example in either mode:

```
mvn compile exec:java  
mvn compile exec:java -Pspring
```

When the example runs, you can use a web browser or REST client like `curl` to call the service. The order ID `123` is hardcoded to return a response, including the token shown in bold:

```
$ curl -i http://localhost:8080/service/order/123  
HTTP/1.1 200 OK  
Content-Length: 37  
Accept: */*  
breadcrumbId: ID-davsclaus-pro-63316-1427025423836-0-7  
id: 123  
Token: 315904563655664740  
User-Agent: curl/7.30.0  
Content-Type: application/json  
Connection: keep-alive  
  
{"id":123, "amount":1, "motor":"Honda"}
```

If you send another request, notice that the token value will change. You can also try to change the example and see what happens if you change the mode to after consumer.

Now you've learned all there is to know about `onCompletion`, which brings us to the next part, where we cover idempotency.

12.5 Idempotency

So far in this chapter, we've talked about how transactions can be used as a means to coordinate state updates among distributed systems to form a unit of work as a whole. Related to this concept is idempotency, the topic for the remainder of this chapter. The term *idempotent* is used in mathematics to describe a function that can be applied multiple times without changing the result beyond the initial result. In computing, *idempotent* is used to describe an operation that will produce the same results if executed once or multiple times.

Idempotency is documented in the EIP book as the Idempotent Consumer pattern. This pattern deals with the problem of how to reliably exchange messages in a distributed system. The book covers the complexity of how a sender can send a message safely to a receiver and many of the situations where something can go wrong. [Figure 12.17](#) shows an example.

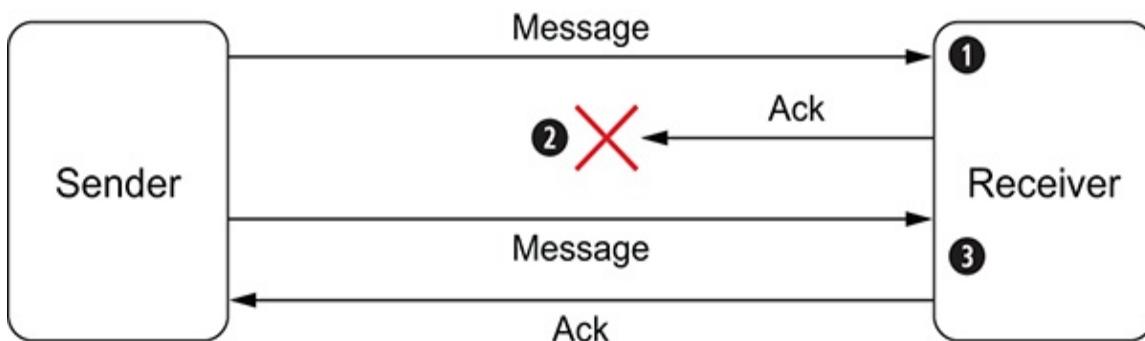


Figure 12.17 The receiver receives duplicate messages because of the problem of returning acknowledgements from the receiver to the sender.

In this example, when a sender sends a message to a receiver ①, the sender can know that the receiver has received and accepted the message only if an acknowledge message is returned from the receiver. But if that acknowledgment is lost due to a networking issue ②, the sender may need to resend the message that the receiver has already received—a duplicate message ③.

Therefore, many messaging systems, such as Apache ActiveMQ, have capabilities to eliminate duplicate messages, so the users don't have to worry about duplicates. In Camel, this

mechanism is implemented as the Idempotent Consumer EIP.

12.5.1 IDEMPOTENT CONSUMER EIP

In Camel, the Idempotent Consumer EIP is used to ensure routing a message once and only once. To achieve this, Camel needs to be able to detect duplicate messages, which involves the following two procedures:

- Generating a unique key for each message
- Storing and retrieving previously seen keys

The heart of the Idempotent Consumer implementation is the *repository*, which is defined as the interface `org.apache.camel.spi.IdempotentRepository` with the following methods:

```
boolean add(E key);
boolean contains(E key);
boolean remove(E key);
boolean confirm(E key);
```

Figure 12.18 illustrates how this works in Camel, and the following steps detail the process:

1. When an Exchange reaches the idempotent consumer, the unique key is evaluated from the Exchange by using the configured Camel Expression. For example, this can be a message header or an XPath expression.
2. The key is checked against the idempotent repository to see whether it's a duplicate. This is done by calling the `add` method. If the method returns `false`, the key was successfully added and no duplicate was detected. If the method returns `true`, an existing key already exists and the message is detected as a duplicate.
3. If the Exchange isn't a duplicate, it's routed as usual.
4. When the Exchange is finished being routed, either the `commit` or `rollback` method of its synchronization is invoked.
5. To commit the Exchange, the `confirm` method is invoked on the

idempotent repository. This allows the idempotent repository to do any post cleanup or other necessary tasks when an Exchange has been committed.

6. To roll back the Exchange, the `remove` method is invoked on the idempotent repository, which removes the key from the repository. Because the key is removed, then upon redelivery of the same message, Camel will have no prior knowledge of having seen this message, and therefore won't regard the message as a duplicate. If you want to prevent this and regard the redelivered message as a duplicate, you can configure the option `removeOnFailure` to `false` on the idempotent repository. Consequently, upon rollback, the key isn't removed but is confirmed (upon rollback, the same action is performed as for a commit—step 5).

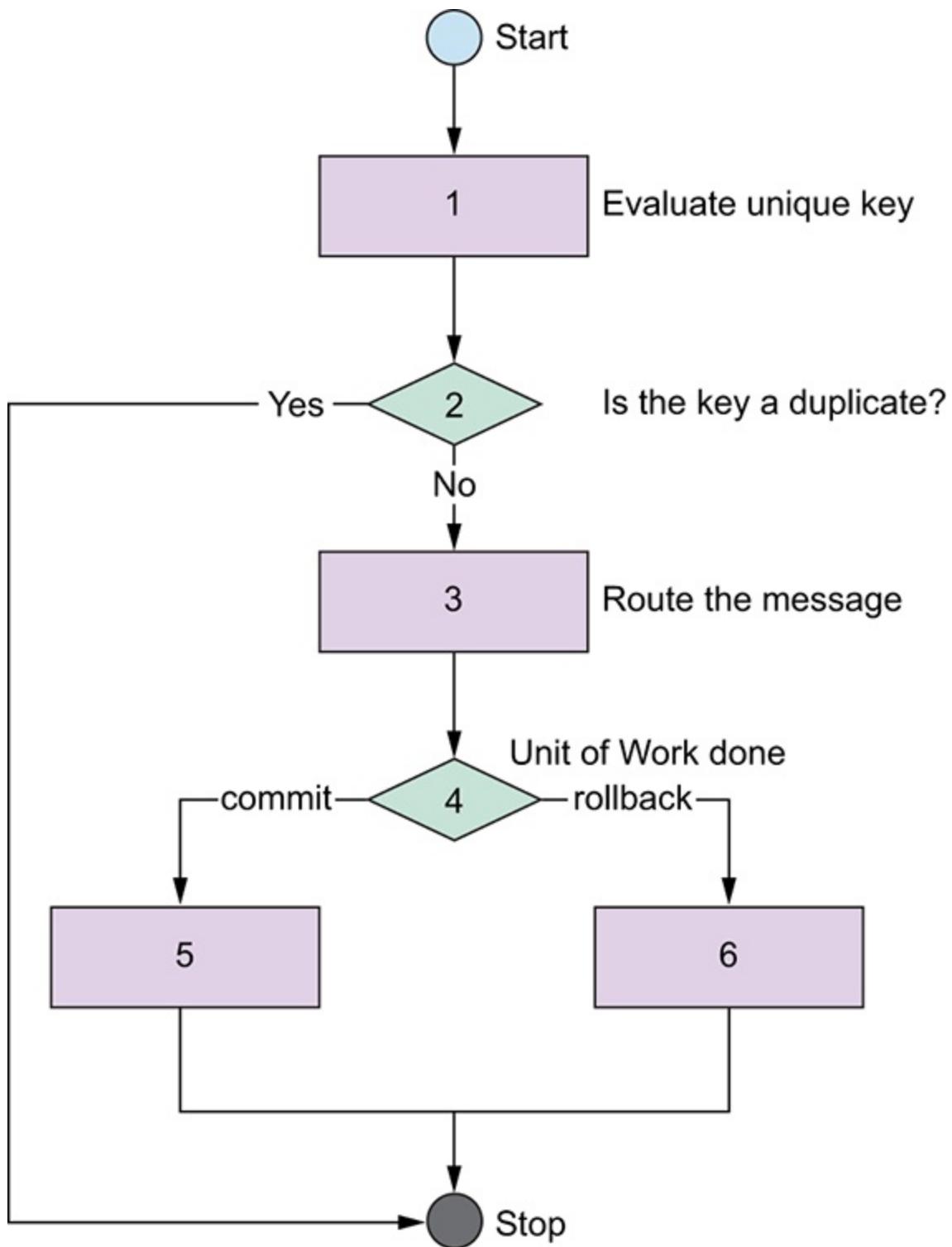


Figure 12.18 How the Idempotent Consumer EIP operates in Camel

Let's see how to do this in action.

USING THE IDEMPOTENT CONSUMER EIP

In order to use the idempotent consumer EIP, you need to set up the idempotent repository first. The following listing shows an example of how to do this using XML DSL.

Listing 12.11 Using Idempotent Consumer EIP in XML DSL

```
<bean id="repo"
```

①

Using the memory idempotent repository

```
class="org.apache.camel.processor.idempotent.MemoryIdempotentRepository"/>
```

```
<camelContext
xmlns="http://camel.apache.org/schema/spring">
<route>
<from uri="seda:inbox"/>
<log message="Incoming order ${header.orderId}" />
<to uri="mock:inbox"/>
<idempotentConsumer
messageIdRepositoryRef="repo">
```

②

Route segment using the idempotent consumer

```
<header>orderId</header>
```

③

Expression to evaluate the unique key

```
<log message="Processing order ${header.orderId}" />
<to uri="mock:order"/>
</idempotentConsumer>
</route>
</camelContext>
```

In this example, you use the built-in memory-based repository,

which is defined as a bean in XML DSL ❶. In the Camel route, you then wrap the routing segment that you want to be invoked only once per unique message ❷. The attribute messageIdRepositoryRef ❸ must be configured to refer to the chosen repository ❶. The example uses the message header orderId as the unique key ❸.

NOTE The unique key is evaluated as a String type. Therefore, you can use only information from the Exchange that can be converted to a String type. You can't use complex types such as Java objects unless they generate unique keys using their `toString` method.

The example from [listing 12.11](#) can be written using Java DSL, as in the following listing.

[Listing 12.12](#) Idempotent Consumer using Java DSL

```
public void configure() throws Exception {  
    IdempotentRepository repo = new  
    MemoryIdempotentRepository() ❶
```

❶ Using the in-memory idempotent repository

```
from("seda:inbox")  
    .log("Incoming order ${header.orderId}")  
    .to("mock:inbox")  
    .idempotentConsumer(header("orderId"), repo) ❷
```

❷

[Route segment using the idempotent consumer, and expression to evaluate unique key](#)

```
    .log("Processing order ${header.orderId}")  
    .to("mock:order")  
    .end();
```

```
}
```

The book's source code contains this example in the chapter12/idempotent directory. You can try the example using the following Maven goals:

```
mvn test -Dtest=SpringIdempotentTest  
mvn test -Dtest=IdempotentTest
```

Running this example will send five messages to the Camel route, two of which will be duplicated, so only three messages will be processed by the idempotent consumer. The example uses the following code to send the five messages:

```
template.sendBodyAndHeader("seda:inbox", "Motor",  
"orderId", "123");  
template.sendBodyAndHeader("seda:inbox", "Motor",  
"orderId", "123");  
template.sendBodyAndHeader("seda:inbox", "Tires",  
"orderId", "789");  
template.sendBodyAndHeader("seda:inbox", "Brake pad",  
"orderId", "456");  
template.sendBodyAndHeader("seda:inbox", "Tires",  
"orderId", "789");
```

This outputs the following to the console:

```
2017-04-12 [- seda://inbox] INFO  route1 - Incoming order  
123  
2017-04-12 [- seda://inbox] INFO  route1 - Processing  
order 123  
2017-04-12 [- seda://inbox] INFO  route1 - Incoming order  
123  
2017-04-12 [- seda://inbox] INFO  route1 - Incoming order  
789  
2017-04-12 [- seda://inbox] INFO  route1 - Processing  
order 789  
2017-04-12 [- seda://inbox] INFO  route1 - Incoming order  
456  
2017-04-12 [- seda://inbox] INFO  route1 - Processing  
order 456  
2017-04-12 [- seda://inbox] INFO  route1 - Incoming order  
789
```

As you can see, the orders are processed only once, indicated by the log "Processing order ...".

The example uses the in-memory idempotent repository, but Camel provides other implementations.

12.5.2 IDEMPOTENT REPOSITORIES

Camel provides many idempotent repositories, each supporting a different level of service. Some are for standalone use in memory only; others support clustered environments. We've listed six of the most commonly used implementations provided by Apache Camel 2.20, with some tips and information about pros and cons next.

MEMORYIDEMPOTENTREPOSITORY

A fast in-memory store that stores entries in a Map structure. All data will be lost when the JVM is stopped. There's no support for clustering. This store is provided in the camel-core module.

FILEIDEMPOTENTREPOSITORY

A file-based store that uses a Map as a cache for fast lookup. All write operations are synchronized, so the store can be a potential bottleneck for high concurrent throughput. Each write operation rewrites the file, which means for large data sets writes can be slower. This store supports active/passive clustering, where the passive cluster is in standby and takes over processing if the master dies. This store is provided in the camel-core module.

JDBCMESSAGEIDREPOSITORY

A store using JDBC to store keys in a SQL database. This implementation uses no caching, so each operation accesses the database. Each idempotent consumer must be associated with a unique name but can reuse the same database table to store keys. High-volume processing can become a performance bottleneck if the database can't keep up. Clustering is supported in both active/passive and active/active mode. This store is provided in the camel-sql module. To use this store, it's required that you set

up the needed SQL table by using the SQL script listed on the camel-sql documentation web page.

JPAMESSAGEIDREPOSITORY

`JpaMessageIdRepository` is similar to `JdbcMessageIdRepository`, but uses JPA as the SQL layer. This store is provided in the camel-jpa module. To use this store, JPA can automatically create needed SQL tables.

HAZELCASTIDEMPOTENTREPOSITORY

`HazelcastIdempotentRepository` is a store using the Hazelcast in-memory data grid as a shared Map of the keys across the cluster of JVMs. Hazelcast can be configured to be in-memory only or to write the Map structure to persistent store by using different levels of QoS. This implementation is suitable for clustering. This store is provided in the camel-hazelcast module.

INFINISPANIDEMPOTENTREPOSITORY

Similar to Hazelcast, `InfinispanIdempotentRepository` uses JBoss Infinispan instead as the in-memory data grid. This store is provided in the camel-infinispan module.

There are also idempotent repositories for Cassandra, HBase, MongoDB, and Redis that you can find in the corresponding Camel component. You can also write your own custom implementation by implementing `org.apache.camel.spi.IdempotentRepository`.

Cache eviction

Each implementation of the idempotent repository has a different strategy for cache eviction. In-memory-based implementations have a limited size, with the number of entries they keep in the cache using a Least Recently Used

(LRU) cache. The size of this cache can be configured. When there are more keys to be added to the cache, the least used is discarded first.

Other implementations such as SQL and JPA require you to manually delete unwanted records from the database table. In-memory data grids such as Hazelcast, Infinispan, and JCache provide configurations for setting the eviction strategy.

It's recommended to study the documentation of the technology you're using to ensure that you use the proper eviction strategy.

12.5.3 CLUSTERED IDEMPOTENT REPOSITORY

This section covers a common use-case involving clustering and the Idempotent Consumer pattern. In this use-case, a shared filesystem is used for exchanging data that you want to process as fast as possible by scaling up your system in a cluster of active nodes; each node can pick up incoming files and process them.

The file component from Apache Camel has built-in support for idempotency. But for historical reasons, this implementation doesn't support atomic operations that a clustered solution would require; there's a small window of opportunity in which duplicates could still happen. [Figure 12.19](#) illustrates this principle.

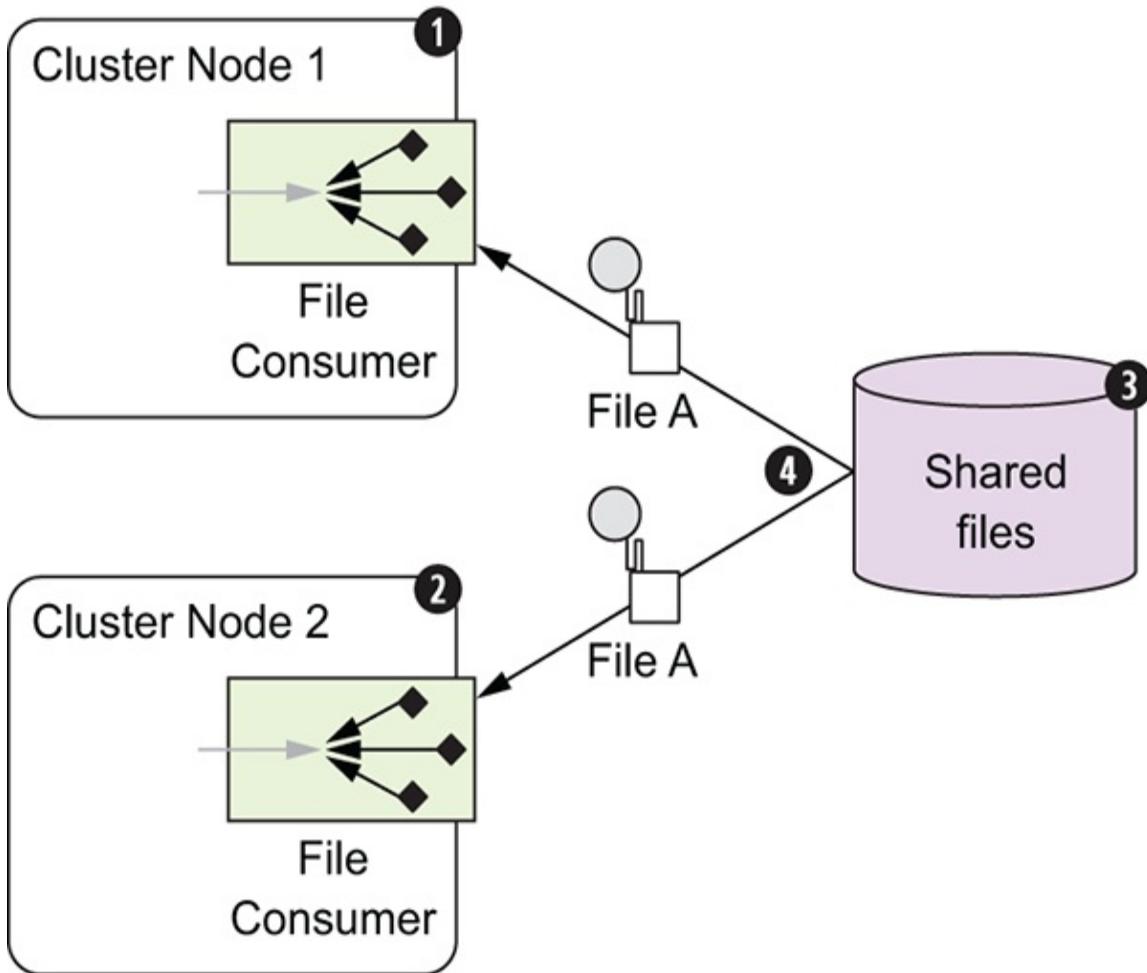


Figure 12.19 Two clustered nodes ① ② with the same Camel application compete concurrently to pick up new files from a shared filesystem ③, and they may pick up the same file ④, which causes duplicates.

When a new file is written to the shared filesystem 3, each active node 1 2 in the cluster reacts independently. Depending on timing and nondeterministic factors, the same file can potentially be picked up by one or more nodes 4. This isn't what you want to happen. You want only one node to process a given file at any time, but you also want each node to process different files at the same time, to achieve higher throughput and distribute the work across the nodes in the cluster, so node 1 can process file A while node 2 processes file B, and so forth.

What you need is something stronger. Yeah, maybe take a break. Grab a strong drink, or if you're in the office, a cup of coffee or tea.

The solution you're looking for is a clustered idempotent repository that the nodes use to coordinate and grant locks to exactly one node. [Figure 12.20](#) shows how this plays out in Camel.

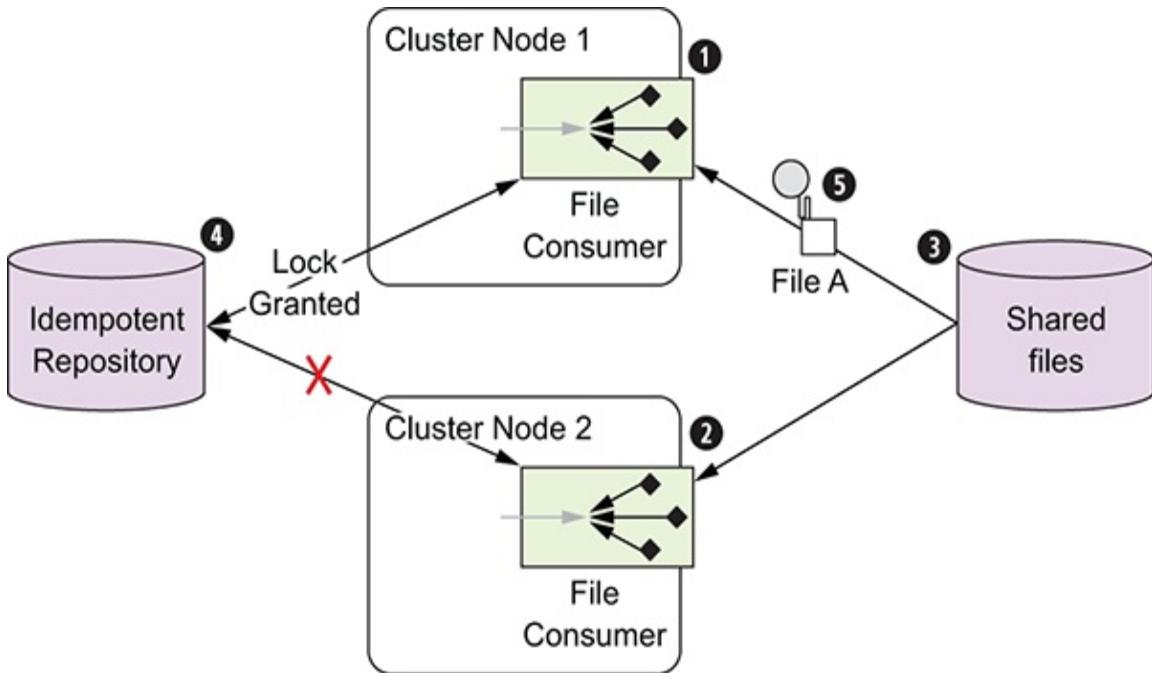


Figure 12.20 Two clustered nodes ①② with the same Camel application compete concurrently to pick up new files from a shared filesystem ③. Both nodes try to acquire a read lock ④ that will permit exactly one with permission to pick up and process the file ⑤. The other nodes will fail to acquire the lock and will skip attempting to pick up and process the file.

Each node in the cluster ① ②, as before, reacts when new files are available from the shared filesystem ③. When a node detects a file to pick up, it now first attempts to grab an exclusive read lock ④ from the clustered idempotent repository. If granted, the file consumer can pick up and process the file ⑤. And when it's done with the file, the read lock is released. While the read lock is granted to a node, any other node that attempts to acquire a read lock for the same filename would be denied by the clustered idempotent repository. The read lock is per file, so each node can process different files concurrently.

Let's see how to do this in Camel using the camel-hazelcast module.

USING HAZELCAST AS CLUSTERED IDEMPOTENT REPOSITORY

The camel-hazelcast module provides a clustered idempotent repository with the class name `HazelcastIdempotentRepository`. To use this repository, you first need to configure Hazelcast, which you can either use as a client to an existing external Hazelcast cluster or embed Hazelcast together with your Camel application.

Hazelcast can be configured by using Java code or from an XML file. This example uses the latter. We've copied the sample config that's distributed from Hazelcast and located the file in `src/main/resources/hazelcast.xml`. Hazelcast allows each node to autojoin using multicast. By doing this, we don't have to manually configure hostnames, IP addresses, and so on for each node to be part of the cluster. We had to change the `hazelcast.xml` configuration from using UDP to TCP with `localhost` to make the example run out of the box on personal computers, as not all popular operating systems have UDP enabled out of the box. We've left code comments in the `hazelcast.xml` file indicating how we did that.

The book's source code contains this example in the `chapter12/hazelcast` directory. To represent two nodes, we have two almost identical source files, `camelinaction.ServerFoo` and `camelinaction.ServerBar`. The following listing shows the source code for `ServerFoo`.

Listing 12.13 A standalone Java Camel application with Hazelcast embedded

```
public class ServerFoo {  
    private Main main;  
  
    public static void main(String[] args) throws Exception {  
        ServerFoo foo = new ServerFoo();  
        foo.boot();    ①
```

①

ServerFoo is a standalone Java application with a main method to boot the application.

```
}  
  
public void boot() throws Exception {  
    HazelcastInstance hz =  
    Hazelcast.newHazelcastInstance(); ②
```

②

Creates and embeds a Hazelcast instance in server mode

```
HazelcastIdempotentRepository repo =  
    new HazelcastIdempotentRepository(hz, "camel"); ③
```

③

Sets up Camel Hazelcast idempotent repository using the name camel

```
main = new Main();  
main.enableHangupSupport();  
main.bind("myRepo", repo); ④
```

④

Binds the idempotent repository to the Camel registry with the name myRepo

```
main.addRouteBuilder(new FileConsumerRoute("FOO",  
100)); ⑤
```

⑤

Adds a Camel route from Java code and runs the application

```
    main.run();  
}
```

Each node is a standalone Java application with a main method
①. Hazelcast is configured using the XML file

src/main/resources/hazelcast.xml. Then you embed and create a Hazelcast instance ②, which by default reads its configuration from the root classpath as the name hazelcast.xml. At this point, Hazelcast is starting up and therefore is ready to use the Hazelcast idempotent repository created ③. The remainder of the code uses Camel Main to easily configure (④ ⑤) and embed a Camel application with the route.

The Camel route that uses the idempotent repository is shown in the following listing.

Listing 12.14 Camel route using clustered idempotent repository with the file consumer

```
public class FileConsumerRoute extends RouteBuilder {  
    public void configure() throws Exception {  
        from("file:target/inbox" +  
            "?delete=true" +  
            "&readLock=idempotent" + ①
```

①

The read lock must be configured as idempotent.

```
"&readLockLoggingLevel=WARN" +  
"&idempotentRepository=#myRepo") ②
```

②

The idempotent repository to use

```
.log(name + " - Received file: ${file:name}")  
.delay(delay) ③
```

③

Delays processing the file so we humans have a chance to see what happens

```
.log(name + " - Done file:      ${file:name}")  
.to("file:target/outbox");  
}  
}
```

The Camel route is a file consumer that picks up files from the target/inbox directory (the shared directory, in this example). The consumer is configured to use idempotent ❶ as its read lock. The idempotent repository is configured with the value #myRepo ❷, which is the name used in [listing 12.13](#). When a file is being processed, it's logged and delayed for a little bit, so the application runs a bit slower and you can better see what happens.

You can try this example located in the chapter12/hazelcast directory by running the following Maven goals for separate shells so they run at the same time. (You can also run them from your IDE, as it's a standard Java main application. The IDE typically supports this by having a right-click menu to run Java applications):

```
mvn compile exec:java -Pfoo  
mvn compile exec:java -Pbar
```

When you start the second application, Hazelcast should log the cluster state:

```
Members [2] {  
    Member [localhost]:5701  
    Member [localhost]:5702 this  
}
```

Here you can see that there are two members in the cluster, linked by using TCP via ports 5701 and 5702.

If you copy a bunch of files to the target/inbox directory, each node will compete concurrently to pick up and process the files. When a node can't acquire an exclusive read lock, a `WARN` is logged:

```
WARN tentRepositoryReadLockStrategy - Cannot acquire read  
lock. Will skip the file: GenericFile[AsyncCallback.java]  
WARN tentRepositoryReadLockStrategy - Cannot acquire read  
lock. Will skip the file: GenericFile[Attachments.java]
```

If you look at the output from both consoles, you should see that the `WARN` from one node isn't a `WARN` from the other node, and vice

versa. You can reduce the number of read-lock contention by shuffling the files in random order. This can be done by configuring the file endpoint with `shuffle=true`. For example, when we tried this, processing 126 files went from 59 WARNS to only 3 when shuffle was turned on.

TIP You can also use the camel-infinispan module instead of Hazelcast, as it offers similar clustering caching capabilities.

One last thing that needs to be configured is the eviction strategy.

CONFIGURING AN EVICTION STRATEGY

Hazelcast by default keeps each element in its distributed cache forever, but this often isn't what you want. For example, in the future you may want to be able to pick up files that have the same names as previously processed files. Or if you're using unique filenames, keeping old files in the cache is a waste of memory. To remove these old filenames from the cache, an eviction strategy must be configured. Hazelcast has many options for this, and in this example we've configured the files to be in the cache for 60 seconds using the time-to-live option.

```
<time-to-live-seconds>60</time-to-live-seconds>
```

Speaking of time, it's time to end this chapter.

12.6 Summary and best practices

Transactions play a crucial role when grouping distinct events together so that they act as a single, coherent, atomic event. In this chapter, we looked at how transactions work in Camel and discovered that Camel lets Spring orchestrate and manage transactions. Using Spring transactions, Camel lets you use an existing and proven transaction framework that works well with

the most popular application servers, message brokers, and database systems.

Here are the best practices you should take away from this chapter:

- *Use transactions when appropriate*—Transactions can be used by only a limited number of resources, such as JMS and JDBC. Therefore, it makes sense to use transactions only when you can use these kinds of resources. If transactions aren’t applicable, you can consider using your own code to compensate and to work as a rollback mechanism.
- *Local or global transactions*—If your route involves only one transactional resource, use local transactions. They’re simpler and much less resource intensive. Use global transactions only if multiple resources are involved.
- *Configuring global transactions is difficult*—When using global transactions (JTA/XA), be advised that they’re difficult to configure. The configuration is specific to the vendor and application server. Don’t underestimate how difficult the configuration can be.
- *Short transactions*—Avoid building systems in which transactions span across a lot of business logic and are routed to many other systems. Keep your transactions short and simple. Instead of one long transaction, break it up into two or more individual steps and use message queues between them. Then you can use short transactions between those message queues and at the same time benefit from using queues that are a great way of decoupling producers and consumers.
- *Favor using one propagation scope*—Attempt to keep your transaction simple by using only one transactional propagation scope, to let all work be within the same transactional context.
- *Test your transactions*—Build unit and integration tests to ensure that your transactions work as expected.
- *Use synchronization tasks*—When you need custom logic to be

executed as either a commit or rollback, use Camel’s Synchronization as part of the exchange unit of work.

- *Use idempotent consumer*—To avoid processing the same message multiple times, use the Idempotent Consumer EIP pattern. To keep state, Camel offers different idempotent repository implementations. Make sure you understand and use the most appropriate for your use-case.
- *Clustered file consumer*—Make sure you use idempotent read lock with a clustered idempotent repository such as Hazelcast, Infnispan, or JCache when you want to scale out your application and have multiple Camel applications process files from a shared directory.

In chapter 13, we’ll turn our attention to using parallel processing with Camel. You’ll learn to how to improve performance, understand the threading model used in Camel, and more.

13

Parallel processing

This chapter covers

- Camel’s threading model
- Configuring thread pools and thread profiles
- Using concurrency with EIPs
- Handling scalability with Camel
- Writing asynchronous Camel components

Concurrency is another word for *multitasking*, and we multitask all the time in our daily lives. We put the coffee on, and while it brews, we grab the tablet and glance at the morning news while we eagerly wait for the coffee to be ready. Computers are also capable of doing multiple tasks—you may have multiple tabs open in your web browser while your mail application is fetching new email, for example.

Juggling multiple tasks is also common in enterprise systems, such as when you’re processing incoming orders, handling invoices, and doing inventory management, and these demands only grow over time. With concurrency, you can achieve higher performance; by executing in parallel, you can get more work done in less time.

Camel processes multiple messages concurrently in Camel routes, and it uses the concurrency features from Java, so we’ll

first discuss how concurrency works in Java before moving on to how thread pools work and how you define and use them in Camel. The *thread pool* is the mechanism in Java that orchestrates multiple tasks. After we've discussed thread pools, we'll move on to using concurrency with the EIPs.

The last section focuses on achieving high scalability with Camel and using this in your custom components. We take extra care to tell you all the ins and outs of writing custom asynchronous components, as this is a bit complicated to understand at first and do the right way. But it's the price to pay for having Camel achieve high scalability (routing using nonblocking mode).

You can also build highly scalable applications using reactive streams, which is covered separately in chapter 20 (available online).

13.1 Introducing concurrency

As we've mentioned, you can achieve higher performance with concurrency. When performance is limited by the availability of a resource, we say it's *bound* by that resource—CPU bound, I/O bound, database bound, and so on. Integration applications are often I/O bound, waiting for replies to come back from remote servers or for files to load from a disk. This usually means you can achieve higher performance using resources more effectively, such as by keeping CPUs busy doing useful work.

Camel is often used to integrate disparate systems, where data is exchanged over the network. There's often a mix of resources, which are a combination of CPU bound or I/O bound. It's likely you can achieve higher performance using concurrency.

To help explain the world of concurrency, we'll look at an example. Rider Auto Parts has an inventory of all the parts its suppliers currently have in stock. It's vital for any business to have the most accurate and up-to-date information in its central ERP system. Having the information locally in the ERP system

means the business can operate without depending on online integration with their suppliers. This system has been in place for many years and is therefore based on what would be considered a legacy system today. The data is exchanged between stakeholders using files that are transferred via FTP servers.

Figure 13.1 illustrates this business process. Figure 13.2 shows the route of the inventory-updating Camel application from figure 13.1. Listing 13.1 details the application.

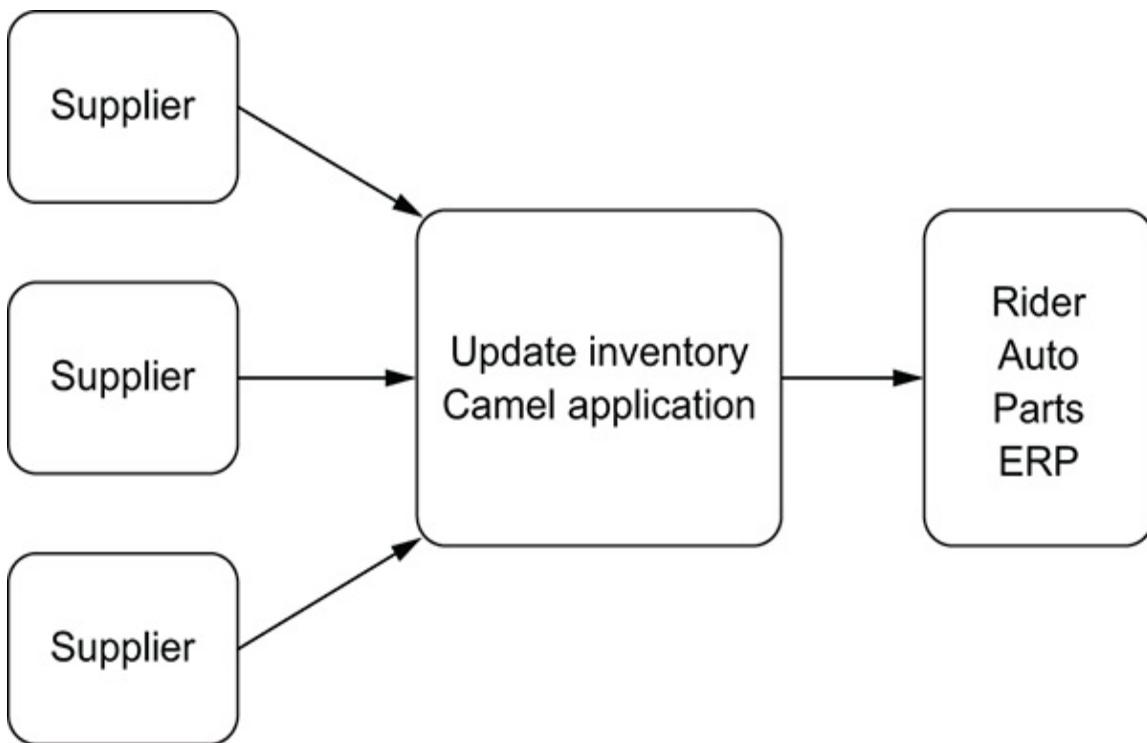


Figure 13.1 Suppliers send inventory updates, which are picked up by a Camel application. The application synchronizes the updates to the ERP system.

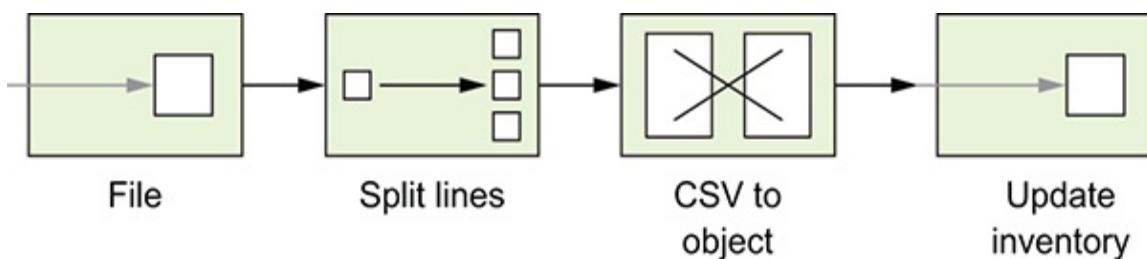


Figure 13.2 A route picks up incoming files, which are split and transformed to be ready for updating the inventory in the ERP system. This application is responsible for loading the files and splitting the file content on a line-by-line basis using the Splitter EIP, which converts the line from CSV format to an

internal object model. The model is then sent to another route that's responsible for updating the ERP system.

Implementing this in Camel is straightforward, as shown in the following listing.

Listing 13.1 Rider Auto Parts application for updating inventory

```
public void configure() throws Exception {  
    from("file:rider/inventory")  
        .log("Starting to process file:  
${header.CamelFileName}")  
        .split(body().tokenize("\n")).streaming() ①
```

1

Splits file line by line

```
        .bean(InventoryService.class, "csvToObject")  
        .to("direct:update")  
    .end()  
    .log("Done processing file:  
${header.CamelFileName}");  
  
    from("direct:update") ②
```

2

Updates ERP system

```
    .bean(InventoryService.class, "updateInventory");  
}
```

Listing 13.1 shows the `configure` method of the Camel `RouteBuilder` that contains the two routes for implementing the application. As you can see, the first route picks up the files and then splits the file content line by line ①. This is done using the Splitter EIP in *streaming mode*. Streaming mode ensures that the entire file isn't loaded into memory; instead it's loaded piece by piece on demand, which ensures low memory usage.

To convert each line from CSV to an object, you use a bean—

the `InventoryService` class. To update the ERP system, you use the `updateInventory` method of the `InventoryService`, as shown in the second route ②.

Now suppose you're testing the application by letting it process a big file with 100,000 lines. If each line takes a tenth of a second to process, processing the file would take 10,000 seconds, which is roughly 167 minutes. That's a long time. In fact, you might end up in a situation where you can't process all the files within the given timeframe.

In a moment, we'll look at various techniques for speeding things up using concurrency. But first let's set up the example to run without concurrency to create a baseline to compare to the concurrent solutions.

13.1.1 RUNNING THE EXAMPLE WITHOUT CONCURRENCY

The book's source code contains this example (both with and without concurrency) in the `chapter13/bigfile` directory.

First, you need a big file to be used for testing. To create a file with 1,000 lines, use the following Maven goal:

```
mvn compile exec:java -PCreateBigFile -Dlines=1000
```

A `bigfile.csv` file will be created in the `target/inventory` directory.

The next step is to start a test that processes `bigfile.csv` without concurrency. This is done using the following Maven goal:

```
mvn test -Dtest=BigFileTest
```

When the test runs, it'll output its progress to the console.

`BigFileTest` simulates updating the inventory by sleeping for a tenth of a second, which means it should complete processing the `bigfile.csv` in approximately 100 seconds (with a 1,000-line file). When the test completes, it should log the total time taken:

```
[ad 0 - file://target/inventory] INFO - Inventory 997 updated
```

```
[ad 0 - file://target/inventory] INFO - Inventory 998
updated
[ad 0 - file://target/inventory] INFO - Inventory 999
updated
[ad 0 - file://target/inventory] INFO - Done processing big
file
Took 102 seconds
```

In the following section, you'll see three solutions to run this test more quickly using concurrency.

13.1.2 USING CONCURRENCY

The application can use concurrency by updating the inventory in parallel. [Figure 13.3](#) shows this principle using the Concurrent Consumers EIP.

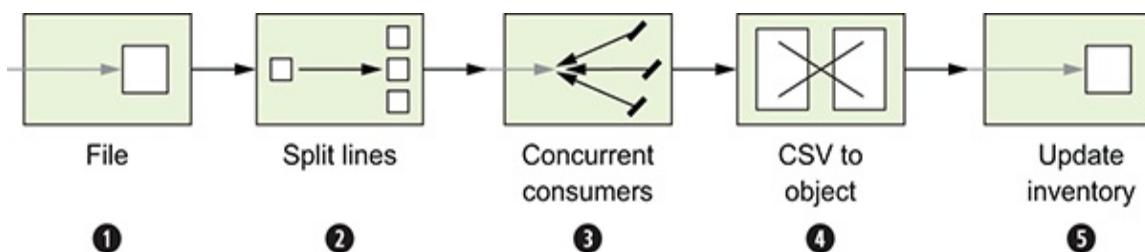


Figure 13.3 Using the Concurrent Consumers EIP to provide concurrency and process inventory updates in parallel

As you can see, the incoming files ① are being split ② into individual lines that are parallelized ③. By doing this, you can parallelize steps ④ and ⑤ in the route. In this example, those two steps could process messages concurrently.

The last step ⑤, which sends messages to the ERP system concurrently, is possible only if the system allows a client to send messages concurrently to it. In some situations, a system won't permit concurrency or may allow up to only a certain number of concurrent messages. Check the service-level agreement (SLA) for the system you integrate with. Another reason to disallow concurrency is if the messages have to be processed in the exact order they're split. Let's try three ways to run the application faster with concurrency:

- Using parallelProcessing options on the Splitter EIP

- Using a custom thread pool on the Splitter EIP
- Using staged event-driven architecture (SEDA)

The first two solutions are features that the Splitter EIP provides out of the box. The last solution is based on the SEDA principle, which uses queues between tasks.

USING PARALLEL PROCESSING

The Splitter EIP offers an option to switch on parallel processing, as shown here:

```
.split(body().tokenize("\n")).streaming().parallelProcessing()
    .bean(InventoryService.class, "csvToObject")
    .to("direct:update")
.end()
```

Configuring this in Spring XML is simple as well:

```
<split streaming="true" parallelProcessing="true">
    <tokenize token="\n"/>
    <bean beanType="camelaction.InventoryService"
        method="csvToObject"/>
    <to uri="direct:update"/>
</split>
```

To run this example, use the following Maven goals:

```
mvn test -Dtest=BigFileParallelTest
mvn test -Dtest=SpringBigFileParallelTest
```

As you'll see, the test is now much faster and completes in about a tenth of the previous time:

```
[Camel Thread 1 - Split] INFO - Inventory 995 updated
[Camel Thread 4 - Split] INFO - Inventory 996 updated
[Camel Thread 9 - Split] INFO - Inventory 997 updated
[Camel Thread 3 - Split] INFO - Inventory 998 updated
[Camel Thread 2 - Split] INFO - Inventory 999 updated
[e://target/inventory?] INFO - Done processing big file
Took 11 seconds
```

When parallelProcessing is enabled, the Splitter EIP uses a

thread pool to process the messages concurrently. The thread pool is, by default, configured to use 10 threads, which helps explain why it's about 10 times faster: the application is mostly I/O bound (reading files and remotely communicating with the ERP system involves a lot of I/O activity). The test would not be 10 times faster if it were solely CPU bound—for example, if all it did was “crunch numbers.”

NOTE In the console output, you'll see that the thread name is displayed, containing a unique thread number, such as CamelThread4-Split. This thread number is a sequential, unique number assigned to each thread as it's created, in any thread pool. This means if you use a second Splitter EIP, the second splitter will most likely have numbers assigned from 11 upward.

You may have noticed from the previous console output that the lines were processed in order; it ended by updating 995, 996, 997, 998, and 999. This is a coincidence, because the 10 concurrent threads are independent and run at their own pace. The reason they appear in order here is that we simulated the update by delaying the message for a tenth of a second, which means they'll all take approximately the same amount of time. But if you take a closer look in the console output, you'll probably see some interleaved lines, such as with order lines 954 and 953:

```
[Camel Thread 5 - Split] INFO - Inventory 951 updated
[Camel Thread 7 - Split] INFO - Inventory 952 updated
[Camel Thread 8 - Split] INFO - Inventory 954 updated
[Camel Thread 9 - Split] INFO - Inventory 953 updated
```

You now know that `parallelProcessing` will use a default thread pool to achieve concurrency. What if you want to have more control over which thread pool is being used?

USING A CUSTOM THREAD POOL

The Splitter EIP also allows you to use a custom thread pool for concurrency. You can create a thread pool using the `java.util.Executors` factory:

```
ExecutorService threadPool =  
Executors.newCachedThreadPool();
```

The `newCachedThreadPool` method creates a thread pool suitable for executing many small tasks. The pool automatically grows and shrinks on demand.

To use this pool with the Splitter EIP, you need to configure it as shown here:

```
.split(body().tokenize("\n")).streaming().executorService(t  
hreadPool)  
    .bean(InventoryService.class, "csvToObject")  
    .to("direct:update")  
.end()
```

Creating the thread pool using Spring XML is done as follows:

```
<bean id="myPool" class="java.util.concurrent.Executors"  
      factory-method="newCachedThreadPool"/>
```

The Splitter EIP uses the pool by referring to it, using the `executorServiceRef` attribute:

```
<split streaming="true" executorServiceRef="myPool">  
    <tokenize token="\n"/>  
    <bean beanType="camelaction.InventoryService"  
          method="csvToObject"/>  
    <to uri="direct:update"/>  
</split>
```

To run this example, use the following Maven goals:

```
mvn test -Dtest=BigFileCachedThreadPoolTest  
mvn test -Dtest=SpringBigFileCachedThreadPoolTest
```

The test is now much faster and completes within a few seconds:

```
[pool-1-thread-442] INFO - Inventory 971 updated  
[pool-1-thread-443] INFO - Inventory 972 updated  
[pool-1-thread-449] INFO - Inventory 982 updated  
[/target/inventory] INFO - Done processing big file
```

```
Took 2 seconds
```

You may wonder why it's now so fast. The cached thread pool is designed to be aggressive and to spawn new threads on demand. It has no upper bounds and no internal work queue, which means that when a new task is being handed over, it'll create a new thread if there are no available threads in the thread pool.

You may also have noticed the thread name in the console output, which indicates that many threads were created; the output shows thread numbers 442, 443, and 449. Many threads have been created because the Splitter EIP splits the file lines more quickly than the tasks update the inventory. The thread pool receives new tasks at a faster pace than it can execute them; new threads are created to keep up.

This can cause unpredicted side effects in an enterprise system; a high number of newly created threads may impact applications in other areas. That's why it's often desirable to use thread pools with an upper limit for the number of threads.

For example, instead of using the cached thread pool, you could use a fixed thread pool. You can use the same Executors factory to create such a pool:

```
ExecutorService threadPool =  
Executors.newFixedThreadPool(20);
```

Creating a fixed thread pool in Spring XML is done as follows:

```
<bean id="myPool" class="java.util.concurrent.Executors"  
      factory-method="newFixedThreadPool">  
    <constructor-arg index="0" value="20"/>  
</bean>
```

To run this example, use the following Maven goals:

```
mvn test -Dtest=BigFileFixedThreadPoolTest  
mvn test -Dtest=SpringBigFileFixedThreadPoolTest
```

The test is now limited to use 20 threads at most:

```
[pool-1-thread-13] INFO - Inventory 997 updated  
[pool-1-thread-19] INFO - Inventory 998 updated
```

```
[ pool-1-thread-5] INFO - Inventory 999 updated  
[target/inventory] INFO - Done processing big file  
Took 6 seconds
```

As you can see by running this test, you can process the 1,000 lines in about 6 seconds and using only 20 threads. The previous test was faster, as it completed in about 2 seconds, but it used nearly 500 threads (this number can vary on different systems). By increasing the fixed thread pool to a reasonable size, you should be able to reach the same timeframe as with the cached thread pool. For example, running with 50 threads completes in about 3 seconds. You can experiment with different pool sizes.

Now, on to the last concurrency solution: SEDA.

USING SEDA

Staged event-driven architecture, or SEDA, is an architecture design that breaks down a complex application into a set of stages connected by queues. In Camel lingo, that means using internal memory queues to hand over messages between routes.

NOTE The Direct component in Camel is the counterpart to SEDA. Direct is fully synchronized and works like a direct method call invocation.

Figure 13.4 shows how you can use SEDA to implement the example. The first route runs sequentially in a single thread. The second route uses concurrent consumers to process the messages that arrive on the SEDA endpoint, using multiple concurrent threads.

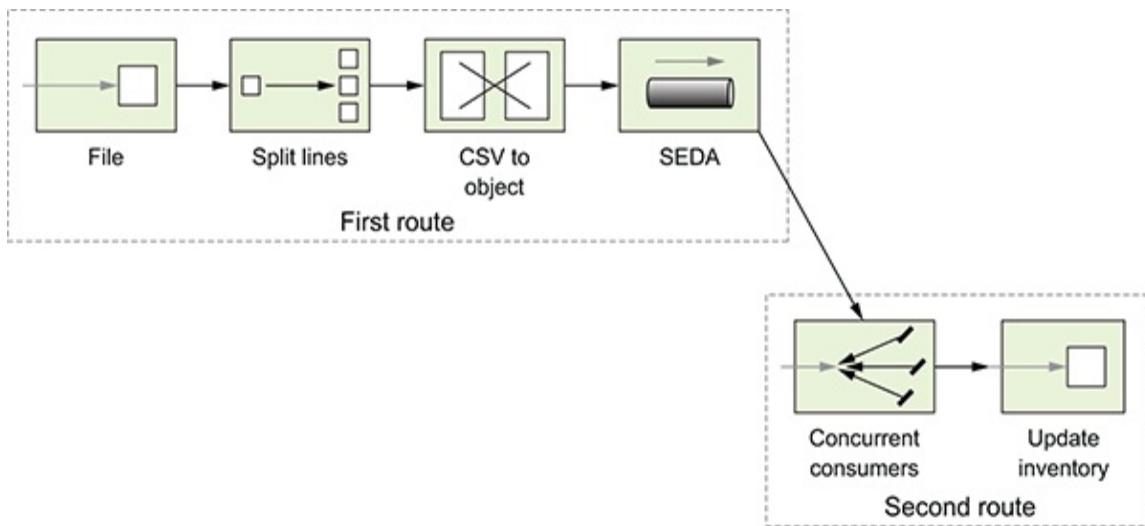


Figure 13.4 Messages pass from the first to the second route using SEDA. Concurrency is used in the second route.

The following listing shows how to implement this solution in Camel using the seda endpoints, shown in bold.

Listing 13.2 Rider Auto Parts inventory-update application using SEDA

```

public void configure() throws Exception {
    from("file:rider/inventory")
        .log("Starting to process file:
${header.CamelFileName}")
        .split(body().tokenize("\n")).streaming()
            .bean(InventoryService.class, "csvToObject")
            .to("seda:update")
        .end()
        .log("Done processing file:
${header.CamelFileName}");

    from("seda:update?concurrentConsumers=20") ①

```

①

SEDA consumers using concurrency

```

        .bean(InventoryService.class, "updateInventory");
}

```

By default, a seda consumer will use only one thread. To use

concurrency, you use the `concurrentConsumers` option to increase the number of threads—to 20 in this listing ①.

The equivalent example from [listing 13.2](#) is as follows when using XML DSL:

```
<route>
    <from uri="file:target/inventory?noop=true"/>
    <log message="Starting to process big file:
    ${header.CamelFileName}"/>
    <split streaming="true">
        <tokenize token="\n"/>
        <bean beanType="camelinaction.InventoryService"
method="csvToObject"/>
        <to uri="seda:update"/>
    </split>
    <log message="Done processing big file:
    ${header.CamelFileName}"/>
</route>

<route>
    <from uri="seda:update?concurrentConsumers=20"/>
```

①

SEDA consumers using concurrency

```
<bean beanType="camelinaction.InventoryService"
      method="updateInventory"/>
</route>
```

To run this example, use the following Maven goals:

```
mvn test -Dtest=BigFileSedaTest
mvn test -Dtest=SpringBigFileSedaTest
```

The test is fast and completes in about 6 seconds:

```
[ead 20 - seda://update] INFO - Inventory 997 updated
[ead 18 - seda://update] INFO - Inventory 998 updated
[read 9 - seda://update] INFO - Inventory 999 updated
Took 6 seconds
```

As you can see from the console output, you're now using 20 concurrent threads to process the inventory update. For example, the last three thread numbers from the output are 20,

18, and 9.

NOTE When using concurrentConsumers with SEDA endpoints, the thread pool uses a fixed size, which means that a fixed number of active threads are waiting at all times to process incoming messages. That's why it's best to use the concurrency features provided by the EIPs, such as the parallelProcessing on the Splitter EIP. It'll use a thread pool that can grow and shrink on demand, so it won't consume as many resources as a SEDA endpoint will.

We've now covered three solutions for applying concurrency to an existing application, and they all greatly improve performance. We were able to reduce the 102-second processing time down to 3 to 7 seconds, using a reasonable size for the thread pool.

TIP The camel-disruptor component works like the SEDA component but uses a different thread model with the LMAX Disruptor library instead of using thread pools from the JDK. You can find more information on the Camel website:
<http://camel.apache.org/disruptor>.

In the next section, we'll review thread pools in more detail and learn about the threading model used in Camel. With this knowledge, you can go even further with concurrency.

13.2 Using thread pools

Using thread pools is common when using concurrency. In fact, thread pools were used in the example in the previous section. It was a thread pool that allowed the Splitter EIP to work in parallel and speed up the performance of the application.

In this section, we'll start from the top and briefly recap what a thread pool is and how it's represented in Java. Then we'll show you the default thread pool profile used by Camel and how to create custom thread pools using Java DSL and XML DSL.

13.2.1 UNDERSTANDING THREAD POOLS IN JAVA

A *thread pool* is a group of threads created to execute a number of tasks in a task queue. Figure 13.5 shows this principle.

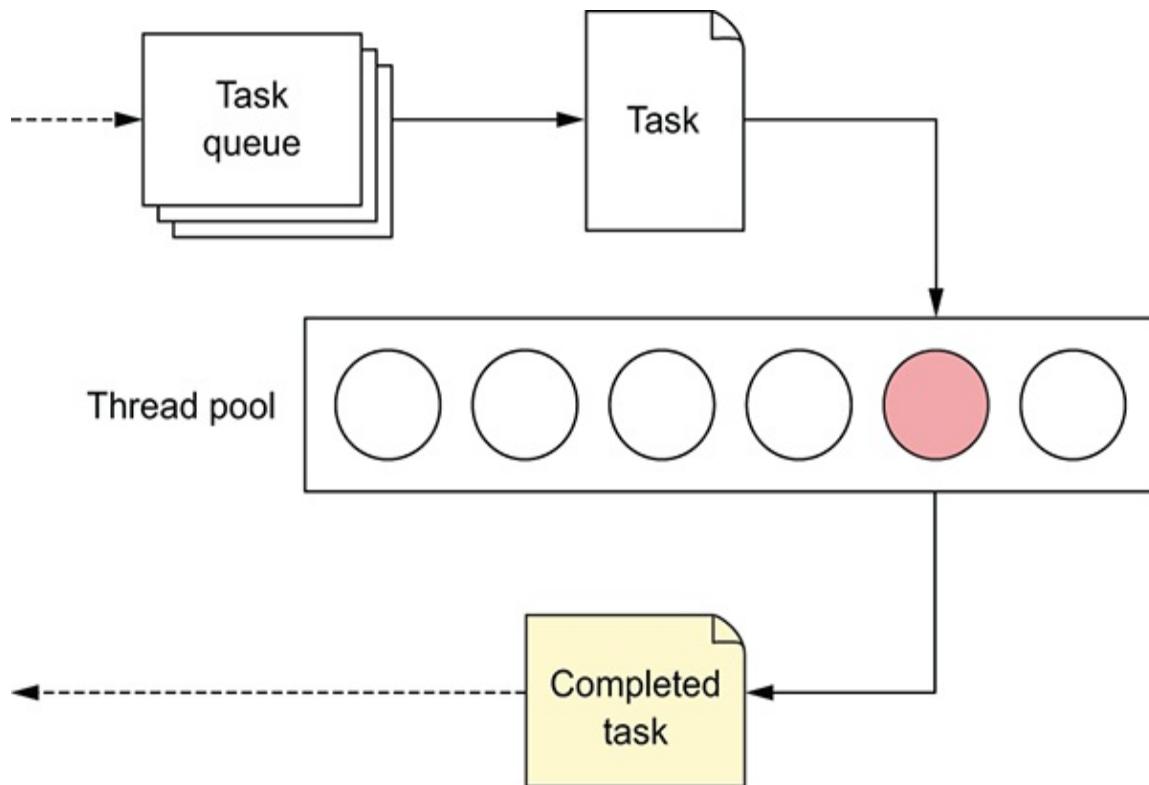


Figure 13.5 Tasks from the task queue wait to be executed by a thread from the thread pool.

NOTE For more info on the Thread Pool pattern, see the Wikipedia article on the subject:
http://en.wikipedia.org/wiki/Thread_pool.

Thread pools were introduced in Java 1.5 by the new concurrency API residing in the `java.util.concurrent` package.

In the concurrency API, the `ExecutorService` interface is the client API that you use to submit tasks for execution. Clients of this API are both Camel end users and Camel itself, because Camel fully uses the concurrency API from Java.

NOTE Readers already familiar with Java's concurrency API may be in familiar waters as we go further in this chapter. If you want to learn in more depth about the Java concurrency API, we highly recommend *Java Concurrency in Practice* by Brian Goetz (Addison-Wesley Professional, 2006).

In Java, the `ThreadPoolExecutor` class is the implementation of the `ExecutorService` interface, and it provides a thread pool with the options listed in table 13.1.

Table 13.1 Options provided by thread pools from Java

Option	Type	Description
<code>corePoolSize</code>	<code>int</code>	Specifies the number of threads to keep in the pool, even if they're idle.
<code>maximumPoolSize</code>	<code>int</code>	Specifies the maximum number of threads to keep in the pool.
<code>keepAliveTime</code>	<code>long</code>	Sets the idle time for excess threads to wait before they're discarded.
<code>unit</code>	<code>TimeUnit</code>	Specifies the time unit used for the <code>keepAliveTime</code> option.
<code>allowCoreThreadTimeOut</code>	<code>boolean</code>	Determines whether the core threads may time out and terminate if no tasks arrive within the keep alive time.
<code>rejected</code>	<code>RejectedExecution-Handler</code>	Identifies a handler to use when execution is blocked because the thread pool is exhausted.
<code>workQueue</code>	<code>BlockingQueue</code>	Identifies the task queue for holding waiting tasks before they're executed.

threadFactory	ThreadFactory	Specifies a factory to use when a new thread is created.
---------------	---------------	--

As you can see in table 13.1, you can use many options when creating thread pools in Java. To make it easier to create commonly used types of pools, Java provides `java.util.concurrent.Executors` as a factory, which you saw in section 13.1.2. In section 13.2.3, you'll see how Camel makes creating thread pools even easier.

When working with thread pools, you often must deal with additional tasks. For example, it's important to ensure that the thread pool is shut down when your application is being shut down; otherwise, it can lead to memory leaks. This is particularly important in server environments when running multiple applications in the same server container, such as a Java servlet, Java EE, or OSGi container.

When using Camel to create thread pools, the activities listed in table 13.2 are taken care of out of the box by Camel.

Table 13.2 Activities for managing thread pools taken care of by Camel

Activity	Description
Shutdown	Ensures the thread pool will be properly shut down, which happens when Camel shuts down.
Management	Registers the thread pool in JMX, which allows you to manage the thread pool at runtime. We'll look at management in chapter 16.
Unique thread names	Ensures the created threads will use unique and human-readable names.
Activity logging	Logs lifecycle activity of the pool.

Another good practice that's often neglected is to use human-understandable thread names, because those names are logged in production logs. By allowing Camel to use a common naming standard to name the threads, you can better understand what happens when looking at log files (particularly if your application

is running together with other frameworks that create their own threads). For example, this log entry indicates it's a thread from the Camel file component:

```
[Camel (camel-1) thread #7 - file://riders/inbox] DEBUG -  
Total 3 files to consume
```

If Camel didn't do this, the thread name would be generic and wouldn't give any hint that it's from Camel, nor that it's the file component:

```
[Thread 0] DEBUG - Total 3 files to consume
```

TIP Camel uses a customizable pattern for naming threads. The default pattern is `Camel Thread ##counter# - #name#`. A custom pattern can be configured using `ExecutorServiceManager`.

We'll cover the options listed in table 13.1 in more detail in the next section, when we review the default thread profile used by Camel.

13.2.2 USING CAMEL THREAD POOL PROFILES

Camel thread pools aren't created and configured directly, but via the configuration of thread pool profiles. A *thread pool profile* dictates how a thread pool should be created, based on a selection of the options listed previously in table 13.1.

Thread pool profiles are organized in a simple two-layer hierarchy with custom and default profiles. There's always one default profile, and you can optionally have multiple custom profiles. The default profile is defined using the options listed in table 13.3.

Table 13.3 Settings for the default thread pool profile

Option	Default value	Description
<code>poolSize</code>	10	The thread pool will always contain at least 10 threads in

		the pool.
maxPoolSize	20	The thread pool can grow up to at most 20 threads.
keepAliveTime	false	Determines whether core thread is also allowed to terminate when there are no pending tasks.
allowCoreThreadTimeOut	60	Idle threads are kept alive for 60 seconds, after which they're terminated.
maxQueueSize	1000	The task queue can contain up to 1,000 tasks before the pool is exhausted.
rejectedPolicy	CallerRuns	If the pool is exhausted, the caller thread will execute the task.

As you can see from these default values, the default thread pool can use from 10 to 20 threads to execute tasks concurrently. The rejectedPolicy option corresponds to the rejected option from table 13.1, and it's an enum type allowing four values: Abort, CallerRuns, DiscardOldest, and Discard. The CallerRuns option uses the caller thread to execute the task itself. The other three options either abort by throwing an exception or discard an existing task from the task queue to make room for the new task.

If the maxQueueSize option is configured to 0, there's no task queue in use. When a task is submitted to the thread pool, the task is processed only if there's an available thread in the pool. If there's no free thread, the rejection policy decides what happens (for example, to use the caller thread or fail with an exception). This is demonstrated later in section 13.3.1, where we process files concurrently without a task queue in use. No one-size-fits-all solution exists for every Camel application, so you may have to tweak the default profile values. But usually you're better off leaving the default values alone. Only by load testing your applications can you determine that tweaking the values will produce better results.

CONFIGURING THE DEFAULT THREAD POOL PROFILE

You can configure the default thread pool profile from either

Java or XML DSL.

In Java, you access `ThreadPoolProfile` starting from `CamelContext`. The following code shows how to change the maximum pool size to 50:

```
ExecutorServiceManager manager =  
context.getExecutorServiceManager();  
ThreadPoolProfile profile =  
manager.getDefaultThreadPoolProfile();  
profile.setMaxPoolSize(50);
```

The default `ThreadPoolProfile` is accessible from `ExecutorServiceManager`, which is an abstraction in Camel allowing you to plug in different thread pool providers. We cover `ExecutorServiceManager` in more detail in section 13.2.4.

In XML DSL, you configure the default thread pool profile using the `<threadPoolProfile>` tag:

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
    <threadPoolProfile id="myDefaultProfile"  
                      defaultProfile="true"  
                      maxPoolSize="50"/>  
  
    ...  
</camelContext>
```

It's important to set the `defaultProfile` attribute to `true` to tell Camel that this is the default profile. You can add options if you want to override any of the other options from table [13.3](#).

In some situations, one profile isn't sufficient. You can also define custom profiles.

CONFIGURING CUSTOM THREAD POOL PROFILES

Defining custom thread pool profiles is much like configuring the default profile.

In Java DSL, a custom profile is created using the `ThreadPoolProfileBuilder` class:

```
ThreadPoolProfile custom = new
```

```
ThreadPoolProfileBuilder("bigPool")
    .maxPoolSize(200).build();
context.getExecutorServiceManager().registerThreadPoolProfile(custom);
```

This example increases the maximum pool size to 200. All other options will be inherited from the default profile, which means it will use the default values listed in table 13.3; for example, `keepAliveTime` will be 60 seconds. Notice that this custom profile is given the name `bigPool`; you can refer to the profile in the Camel routes using `executorServiceRef`:

```
.split(body().tokenize("\n")).streaming().executorServiceRef("bigPool")
    .bean(InventoryService.class, "csvToObject")
    .to("direct:update")
.end()
```

And in XML DSL:

```
<split streaming="true" executorServiceRef="bigPool">
    <tokenize token="\n"/>
    <bean beanType="camelaction.InventoryService"
method="csvToObject"/>
    <to uri="direct:update"/>
</split>
```

When Camel creates this route with the Splitter EIP, it refers to a thread pool with the name `bigPool`. Camel will now look in the registry for an `ExecutorService` type registered with the ID `bigPool`. If none is found, it will fall back and see whether there's a known thread pool profile with the ID `bigPool`. And because such a profile has been registered, Camel will use the profile to create a new thread pool to be used by the Splitter EIP. All of this means that `executorServiceRef` supports using thread pool profiles to create the desired thread pools.

When using XML DSL, it's simpler to define custom thread pool profiles. All you have to do is use the `<threadPoolProfile>` tag:

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

```
<threadPoolProfile id="bigPool" maxPoolSize="200"/>
</camelContext>
```

Besides using thread pool profiles, you can create thread pools in other ways. For example, you may need to create custom thread pools if you're using a third-party library that requires you to provide a thread pool. Or you may need to create one, as we did in section 13.1.2, to use concurrency with the Splitter EIP.

13.2.3 CREATING CUSTOM THREAD POOLS

Creating thread pools with the Java API is a bit cumbersome, so Camel provides a nice way of doing this in both Java DSL and XML DSL.

CREATING CUSTOM THREAD POOLS IN JAVA DSL

In Java DSL, you use

`org.apache.camel.builder.ThreadPoolBuilder` to create thread pools, as follows:

```
ThreadPoolBuilder builder = new ThreadPoolBuilder(context);
ExecutorService myPool =
builder.poolSize(5).maxPoolSize(25)
.maxQueueSize(200).build("Cool");
```

`ThreadPoolBuilder` requires `CamelContext` in its constructor, because it'll use the default thread pool profile as the baseline when building custom thread pools. That means `myPool` will use the default value for `keepAliveTime`, which would be 60 seconds.

CREATING CUSTOM THREAD POOLS IN XML DSL

In XML DSL, creating a thread pool is done using the `<threadPool>` tag:

```
<camelContext
xmlns="http://camel.apache.org/schema/spring">
<threadPool id="myPool" threadName="Cool"
            poolSize="5" maxPoolSize="25"
maxQueueSize="200"/>
```

```
<route>
  <from uri="direct:start"/>
  <to uri="log:start"/>
  <threads executorServiceRef="myPool">
```

1

1

Using thread pool in the route

```
    <to uri="log:hello"/>
  </threads>
</route>
</camelContext>
```

As you can see, `<threadPool>` is used inside a `<camelContext>` tag. That's because it needs access to the default thread profile, which is used as baseline (just as the `ThreadPoolBuilder` requires `CamelContext` in its constructor).

The preceding route uses a `<threads>` tag that references the custom thread pool 1. If a message is sent to the `direct:start` endpoint, it should be routed to `<threads>`, which will continue routing the message using the custom thread pool. This can be seen in the console output that logs the thread names. The thread name in the log will use the name you configured in the `<threadPool>` tag, which in the example is `cool`, as shown in bold:

```
Camel (camel-1) thread #0 cool] INFO hello -
Exchange[Body:Hello Camel]
```

NOTE When using `executorServiceRef` to look up a thread pool, Camel will first check for a custom thread pool. If none is found, Camel will fall back and see if a thread pool profile exists with the given name; if so, a new thread pool is created based on that profile.

All thread pool creation is done using `ExecutorServiceManager`, which defines a pluggable API for using thread pool providers.

13.2.4 USING EXECUTORSERVICEMANAGER

The `org.apache.camel.spi.ExecutorServiceManager` interface defines a pluggable API for thread pool providers. Camel will, by default, use the `DefaultExecutorServiceManager` class, which creates thread pools using the concurrency API in Java. When you need to use a different thread pool provider (for example, a provider from a Java EE server), you can create a custom `ExecutorServiceManager` to work with the provider.

In this section, we'll show you how to configure Camel to use a custom `ExecutorServiceManager`, leaving the implementation of the provider up to you.

CONFIGURING CAMEL TO USE A CUSTOM EXECUTORSERVICEMANAGER

In Java, you configure Camel to use a custom `ExecutorServiceManager` via the `setExecutorServiceManager` method on `CamelContext`:

```
CamelContext context = ...
context.setExecutorServiceManager(myExecutorServiceManager)
;
```

In XML DSL, it's easy because all you have to do is define a bean. Camel will automatically detect and use it:

```
<bean id="myExecutorService"
      class="camelinaction.MyExecutorServiceManager"/>
```

So far in this chapter, we've mostly used thread pools in Camel routes, but they're also used in other areas, such as in some Camel components.

USING EXECUTORSERVICEMANAGER IN A CUSTOM COMPONENT

The `ExecutorServiceManager` defines methods for working with thread pools.

Suppose you're developing a custom Camel component and you need to run a scheduled background task. When running a background task, it's recommended that you use the `ScheduledExecutorService` as the thread pool, because it's

capable of executing tasks in a scheduled manner.

Creating the thread pool is easy with the help of Camel's ExecutorServiceManager, as shown in the following listing.

Listing 13.3 Using ExecutorServiceManager to create a thread pool

```
public class MyComponent extends DefaultComponent
implements Runnable {
    private static final Log LOG =
LogFactory.getLog(MyComponent.class);
    private ScheduledExecutorService executor;

    public void run() {
        LOG.info("I run now"); ①
```

①

Runs scheduled task

```
}
```

```
protected void doStart() throws Exception {
    super.doStart();
    executor =
getCamelContext().getExecutorServiceManager()
    .newScheduledThreadPool(this,
        "MyBackgroundTask", 1); ②
```

②

Creates scheduled thread pool

```
executor.scheduleWithFixedDelay(this, 1, 1,
TimeUnit.SECONDS);
}

protected void doStop() throws Exception {

getCamelContext().getExecutorServiceManager().shutdown(executor);
    super.doStop();
}
```

[Listing 13.3](#) illustrates the principle of using a scheduled thread pool to repeatedly execute a background task. The custom component extends `DefaultComponent`, which allows you to override the `doStart` and `doStop` methods to create and shut down the thread pool. In the `doStart` method, you create the `ScheduledExecutorService` using `ExecutorServiceManager` ❷ and schedule it to run the task ❶ once every second using the `scheduleWithFixedDelay` method.

The book's source code contains this example in the `chapter13/pools` directory. You can try it using the following Maven goal:

```
mvn test -Dtest=MyComponentTest
```

When it runs, you'll see the following output in the console:

```
Waiting for 10 seconds before we shutdown
[Camel (camel-1) thread #0 - MyBackgroundTask] INFO
MyComponent - I run now
[Camel (camel-1) thread #0 - MyBackgroundTask] INFO
MyComponent - I run now
```

You now know that thread pools are how Java achieves concurrency; they're used as executors to execute tasks concurrently. You also know how to use this to process messages concurrently in Camel routes, and you saw several ways of creating and defining thread pools in Camel.

When modeling routes in Camel, you'll often use EIPs to build the routes to support your business cases. In section 13.1, you used the Splitter EIP and learned to improve performance using concurrency. In the next section, we'll look at other EIPs you can use with concurrency.

13.3 Parallel processing with EIPs

Some of the EIPs in Camel support parallel processing out of the box—they're listed in table 13.4. This section takes a look at them and the benefits they offer.

Table 13.4 EIPs in Camel that support parallel processing

EIP	Description
Aggregator	The Aggregator EIP allows concurrency when sending out completed and aggregated messages. We covered this pattern in chapter 5.
Delayer	The Delayer EIP allows you to delay messages during routing. The delay can either be synchronous (the current thread is blocked) or asynchronous (the delay uses a scheduled thread pool to continue routing in a future time). Only in the latter situation can the Delayer EIP process messages in parallel due to the usage of the scheduled thread pool.
Multicast	The Multicast EIP allows concurrency when sending a copy of the same message to multiple recipients. We discussed this pattern in chapter 2, and we'll use it in an example in section 13.3.2.
Recipient List	The Recipient List EIP allows concurrency when sending copies of a single message to a dynamic list of recipients. This works in the same way as the Multicast EIP, so what you learned there also applies for this pattern. We covered this pattern in chapter 2.
Splitter	The Splitter EIP allows concurrency when each split message is being processed. You saw how to do this in section 13.1. This pattern was also covered in chapter 5.
Threads	The Threads EIP always uses concurrency to hand over messages to a thread pool that will continue processing the message. You saw an example of this in section 13.2.3, and we'll cover it a bit more in section 13.3.1.
Throttler	The Throttler EIP allows you to throttle messages during routing, where some messages may be held back when the throttling limit has been reached. The throttling can either be <i>synchronous</i> (the current thread is blocked) or

tl er	<i>asynchronous</i> (the throttling uses a scheduled thread pool to continue routing in a future time). Only in the latter situation can the Throttler EIP process messages in parallel due to the usage of the scheduled thread pool. This is similar to how the Delayer EIP works with parallel processing.
W ir e T a p	The Wire Tap EIP allows you to spawn a new message and let it be sent to an endpoint using a new thread while the calling thread can continue to process the original message. The Wire Tap EIP always uses a thread pool to execute the spawned message. This is covered in section 13.3.3. You encountered the Wire Tap pattern in chapter 2.

All the EIPs from table 13.4 can be configured to enable concurrency in the same way. You can turn on `parallelProcessing` to use thread pool profiles to apply a matching thread pool; this is likely what you'll want to use in most cases. Or you can refer to a specific thread pool using the `executorService` option. You've already seen this in action in section 13.1.2, where we used the Splitter EIP.

In the following three sections, we'll see how to use the Threads, Multicast, and Wire Tap EIPs in a concurrent way.

13.3.1 USING CONCURRENCY WITH THE THREADS EIP

The Threads EIP is the only EIP that has additional options in the DSL offering a fine-grained definition of the thread pool to be used. These additional options were listed previously in table 13.3.

For example, the thread pool from section 13.2.3 could be written as follows:

```
<camelContext
xmlns="http://camel.apache.org/schema/spring">
<route>
    <from uri="direct:start"/>
    <to uri="log:start"/>
    <threads threadName="Cool" poolSize="5"
maxPoolSize="25"
                maxQueueSize="200">
        <to uri="log:cool"/>
    </threads>
```

```
</route>  
</camelContext>
```

Figure 13.6 illustrates which threads are in use when a message is being routed using the Threads EIP.

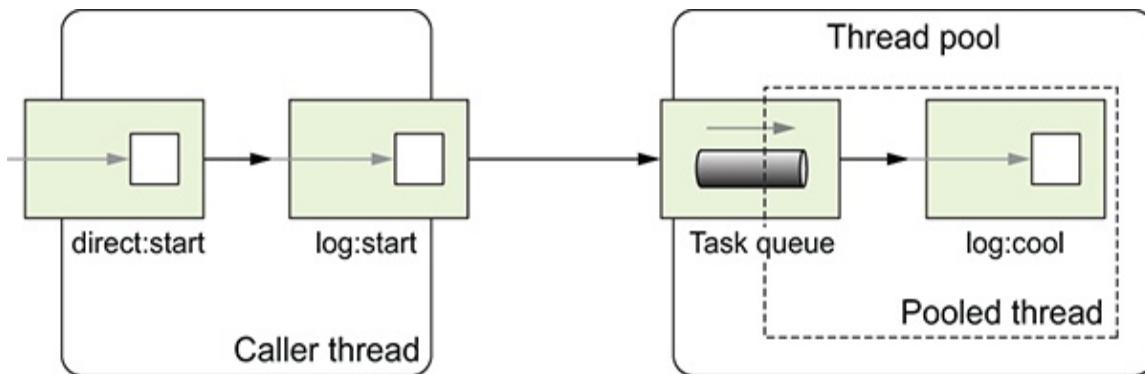


Figure 13.6 Caller and pooled threads are in use when a message is routed. Two threads are active when a message is being routed. The caller thread hands over the message to the thread pool. The thread pool then finds an available thread in its pool to continue routing the message.

You can run this example from the chapter13/pools directory using the following Maven goal:

```
mvn test -Dtest=SpringInlinedThreadPoolTest
```

You'll see the following in the console:

```
[main] INFO start -  
Exchange[Body:Hello Camel]  
Camel (camel-1) thread #0 - Cool] INFO hello -  
Exchange[Body:Hello Camel]
```

The first set of brackets contains the thread name. You see, as expected, two threads in play: `main` is the caller thread, and `cool` is from the thread pool.

PARALLEL PROCESSING FILES

You can use the Threads EIP to achieve concurrency when using Camel components that don't offer concurrency. A good example is the Camel file component, which uses a single thread to scan and pick up files. Using the Threads EIP, you can allow the picked-up files to be processed concurrently.

The following listing shows an example of how to do that with XML DSL.

Listing 13.4 Processing files concurrently with the Thread EIP

```
<bean id="delayProcessor"
      class="camelinaction.DelayProcessor"/>

<camelContext
  xmlns="http://camel.apache.org/schema/spring">

  <route id="myRoute" autoStartup="false">
    <from uri="file:target/inbox?delete=true"/> 1

```

1

Consuming files from the starting directory

```
    <log message="About to process ${file:name} thread
#${threadName}"/>
    <threads poolSize="10" maxQueueSize="0"> 2

```

2

Using Threads EIP to process the files in parallel

```
      <log message="Start ${file:name} thread
#${threadName}"/>
      <process ref="delayProcessor"/> 3

```

3

Random delay processing the files to simulate CPU processing

```
        <log message="Done ${file:name} thread
#${threadName}"/>
        </threads>
        <to uri="log:done?groupSize=10"/> 4

```

4

Log average processing speed per 10 files

```
</route>
```

```
</camelContext>
```

This route starts from a file directory ❶, where all incoming files are picked up by the Camel file component. This is done using a single thread. To archive parallel processing of the files, you use threads ❷ configured with a thread pool of 10 active threads and no task queue. By setting `maxQueueSize` to 0, you don't use any in-memory task queue in the thread pool. The file consumer will pick up only new files, when there's a thread available from Threads EIP to process the file. By default, the thread pool would otherwise have a `maxQueueSize` of 1000 as outlined previously in table 13.3. A processor ❸ is used to delay processing each file—otherwise, the files are all processed too fast. A log endpoint ❹ is used to log the average processing speed per 10 files.

The book's source code contains this example in `chapter13/pools`, and you can run the example using the following Maven goals:

```
mvn test -Dtest=FileThreadsTest  
mvn test -Dtest=SpringFileThreadsTest
```

We encourage you to try this example and experiment with the various settings of the Threads EIP. In addition, pay attention to the logging output to see which thread is doing what.

Let's look at how Rider Auto Parts improves performance using concurrency with the Multicast EIP.

13.3.2 USING CONCURRENCY WITH THE MULTICAST EIP

Rider Auto Parts has a web portal where its employees can look up information, such as the current status of customer orders. When selecting a particular order, the portal needs to retrieve information from three systems to gather an overview of the order. Figure 13.7 illustrates this.

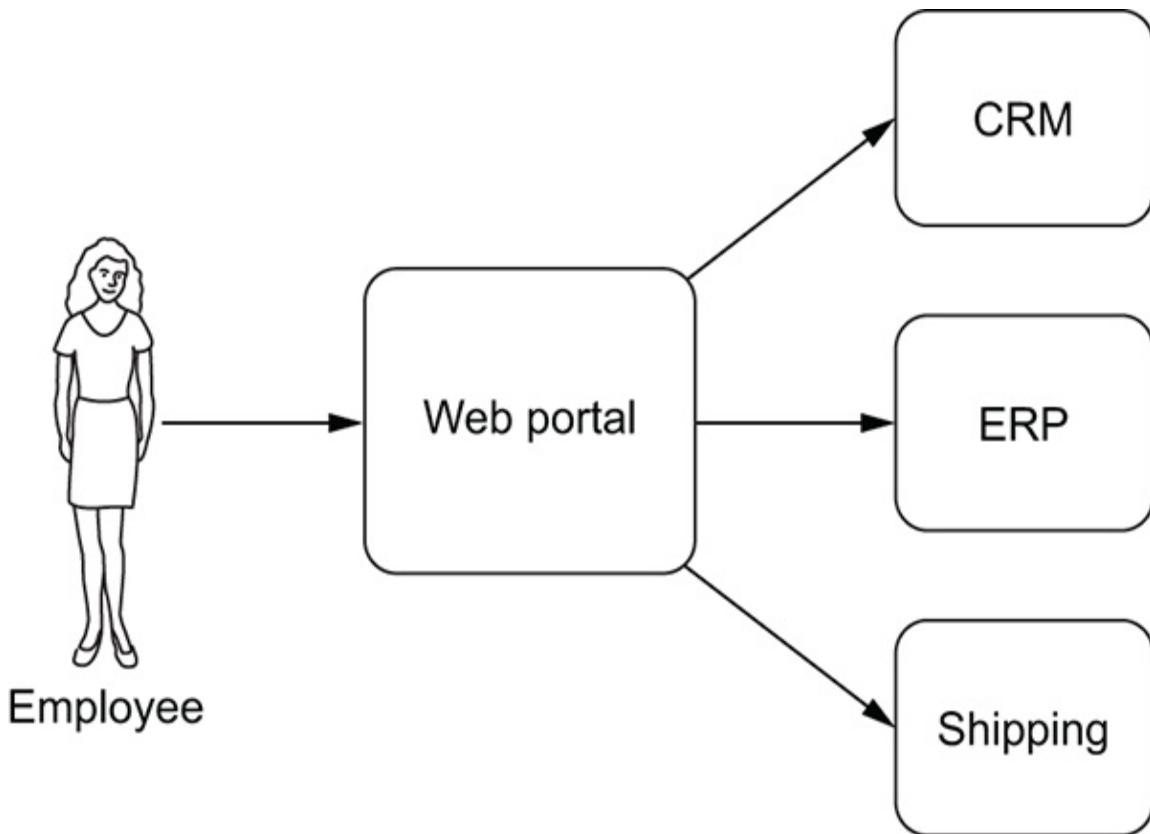


Figure 13.7 The web portal gathers information from three systems to compile the overview that's presented to the employee.

Your boss has summoned you to help with this portal. The employees have started to complain about poor performance, and it doesn't take you more than an hour to find out why: the portal retrieves the data from the three sources in sequence. This is obviously a good use-case for using concurrency to improve performance.

You also look in the production logs and see that a single overview takes 4.0 seconds ($1.4 + 1.1 + 1.5$ seconds) to complete. You tell your boss that you can improve the performance by gathering the data in parallel.

Back at your desk, you build a portal prototype in Camel that resembles the current implementation. The prototype uses the Multicast EIP to retrieve data from the three external systems as follows:

```
<route>
```

```

<from uri="direct:portal"/>
<multicast strategyRef="aggregatedData">
    <to uri="direct:crm"/>
    <to uri="direct:erp"/>
    <to uri="direct:shipping"/>
</multicast>
<bean ref="combineData"/>
</route>

```

The Multicast EIP will send copies of a message to the three endpoints and aggregate their replies using the aggregatedData bean. When all data has been aggregated, the combineData bean is used to create the reply that will be displayed in the portal.

You decide to test this route by simulating the three systems using the same response times as from the production logs. Running your test yields the following performance metrics:

```

TIMER - [Message: 123] sent to: direct://crm took: 1404 ms.
TIMER - [Message: 123] sent to: direct://erp took: 1101 ms.
TIMER - [Message: 123] sent to: direct://shipping took:
1501 ms.
TIMER - [Message: 123] sent to: direct://portal took: 4139
ms.

```

As you can see, the total time is 4.1 seconds when running in sequence. Now you enable concurrency with the parallelProcessing options:

```

<route>
    <from uri="direct:portal"/>
    <multicast strategyRef="aggregatedData"
               parallelProcessing="true">
        <to uri="direct:crm"/>
        <to uri="direct:erp"/>
        <to uri="direct:shipping"/>
    </multicast>
    <bean ref="combineData"/>
</route>

```

This gives much better performance:

```

TIMER - [Message: 123] sent to: direct://erp took: 1105 ms.
TIMER - [Message: 123] sent to: direct://crm took: 1402 ms.
TIMER - [Message: 123] sent to: direct://shipping took:
1502 ms.

```

```
TIMER - [Message: 123] sent to: direct://portal took: 1623 ms.
```

The numbers show that response time went from 4.1 to 1.6 seconds, which is an improvement of roughly 250 percent. Note that the logged lines aren't in the same order as the sequential example. With concurrency enabled, the lines are logged in the order that the remote services' replies come in. Without concurrency, the sequential order is always fixed as defined by the Camel route.

The book's source code contains this example in the chapter13/eip directory. You can try the two scenarios using the following Maven goals:

```
mvn test -Dtest=MulticastTest  
mvn test -Dtest=MulticastParallelTest
```

You've now seen how the Multicast EIP can be used concurrently to improve performance. The Aggregator, Recipient List, and Splitter EIPs can be configured with concurrency in the same way as the Multicast EIP.

The next pattern we'll look at using with concurrency is the Wire Tap EIP.

13.3.3 USING CONCURRENCY WITH THE WIRE TAP EIP

The Wire Tap EIP uses a thread pool to process the tapped messages concurrently. You can configure which thread pool it should use, and if no pool has been configured, it'll fall back and create a thread pool based on the default thread pool profile.

Suppose you want to use a custom thread pool when using the Wire Tap EIP. First, you must create the thread pool to be used, and then you pass that in as a reference to the wire tap in the route:

```
public void configure() throws Exception {  
    ExecutorService lowPool = new  
    ThreadPoolBuilder(context)
```

```

    .poolSize(1).maxPoolSize(5).build("LowPool");

    from("direct:start")
        .log("Incoming message ${body}")
        .wireTap("direct:tap", lowPool)
        .to("mock:result");

    from("direct:tap")
        .log("Tapped message ${body}")
        .to("mock:tap");
}

```

The equivalent route in XML DSL is as follows. Notice how <wireTap> uses the attribute named executorServiceRef to refer to the custom thread pool to be used:

```

<camelContext
xmlns="http://camel.apache.org/schema/spring">

    <threadPool id="lowPool"
                poolSize="1" maxPoolSize="5"
threadName="LowPool"/>

    <route>
        <from uri="direct:start"/>
        <log message="Incoming message ${body}" />
        <wireTap uri="direct:tap"
executorServiceRef="lowPool"/>
        <to uri="mock:result"/>
    </route>

    <route>
        <from uri="direct:tap"/>
        <log message="Tapped message ${body}" />
        <to uri="mock:tap"/>
    </route>
</camelContext>

```

The book's source code contains this example in the chapter13/eip directory. You can run the example using the following Maven goals:

```

mvn test -Dtest=WireTapTest
mvn test -Dtest=SpringWireTapTest

```

When you run the example, the console output should indicate

that the tapped message is being processed by a thread from the LowPool thread pool:

```
[main]                                     INFO route1 -  
Incoming message Hello Camel  
[Camel (camel-1) thread #0 - LowPool] INFO route2 - Tapped  
message Hello Camel
```

Wire Tap and stream-based messages

The Wire Tap EIP creates a shallow copy of the message that's being tapped and routes the second message concurrently. If the message body is streaming based (for example, `java.io.InputStream` type), you should consider enabling stream caching that allows concurrent access to the stream. Chapter 15 provides more details about stream caching.

Wire Tap is useable not only for tapping messages during routing. The pattern can also be used when you want to return an early reply to the caller, while Camel is concurrently processing the message.

RETURNING AN EARLY REPLY TO THE CALLER

Consider an example in which a caller invokes a Camel service in a synchronous manner—the caller is blocked while waiting for a reply. In the Camel service, you want to send a reply back to the waiting caller as soon as possible; the reply is an acknowledgment that the input has been received, so `ok` is returned to the caller. In the meantime, Camel continues processing the received message in another thread.

Figure 13.8 illustrates this example in a sequence diagram.

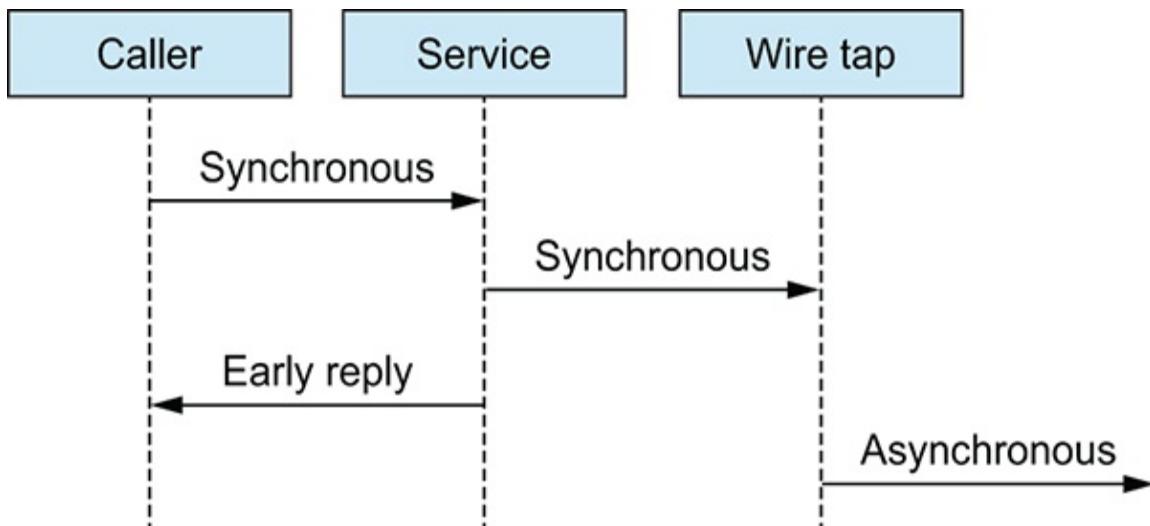


Figure 13.8 A synchronous caller invokes a Camel service. The service lets the wire tap continue processing the message asynchronously while the service returns an early reply to the waiting caller.

The following listing implements this example as a Camel route with the Java DSL.

Listing 13.5 Using Wire Tap to return an early reply message to the caller using Java DSL

```
from("jetty:http://localhost:8080/early").routeId("input")
    .wireTap("direct:incoming") 1
```

1

Taps incoming message

```
    .transform().constant("OK"); 2
```

2

Returns early reply

```
from("direct:incoming").routeId("process") 3
```

3

Route that processes the message asynchronously

```
.convertBodyToString().  
.log("Incoming ${body}")  
.delay(3000)  
.log("Processing done for ${body}")  
.to("mock:result");  
}
```

You use the Wire Tap EIP ❶ to continue routing the incoming message in a separate thread, in the process route ❸. This gives room for the consumer to immediately reply ❷ to the waiting caller.

The next listing shows an equivalent example using XML DSL.

Listing 13.6 Using WireTap to return an early reply message to the caller using XML DSL

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
  
<route routeId="input">  
  <from uri="jetty:http://localhost:8080/early"/>  
  <wireTap uri="direct:incoming"/> ❶
```

❶

Taps incoming message

```
  <transform>  
    <constant>OK</constant> ❷
```

❷

Returns early reply

```
  </transform>  
</route>  
  
<route routeId="process"> ❸
```

❸

Route that processes the message asynchronously

```

<from uri="direct:incoming"/>
<convertBodyTo type="String"/>
<log message="Incoming ${body}"/>
<delay>
    <constant>3000</constant>
</delay>
<log message="Processing done for ${body}"/>
<to uri="mock:result"/>
</route>

</camelContext>

```

The book's source code contains this example in the chapter13/eip directory. You can run the example using the following Maven goals:

```

mvn test -Dtest=EarlyReplyTest
mvn test -Dtest=SpringEarlyReplyTest

```

When you run the example, you should see the console output showing how the message is processed:

```

11:18:15 [main] INFO - Caller calling Camel with message:
Hello Camel
11:18:15 [Camel (camel-1) thread #0 - WireTap] INFO -
Incoming Hello Camel
11:18:15 [main] INFO - Caller finished calling Camel and
received reply: OK
11:18:18 [Camel (camel-1) thread #0 - WireTap] INFO -
Processing done for Hello Camel

```

Notice in the console output that the caller immediately receives a reply within the same second the caller sent the request. The last log line shows that the Wire Tap EIP finished processing the message 3 seconds after the caller received the reply.

NOTE In the preceding example, (the route with ID process) you need to convert the body to a String type to ensure that you can read the message multiple times. This is necessary because Jetty is stream based; consequently, Camel reads the message only once. Or, instead of converting the body, you could enable stream caching (covered in chapter 15).

So far in this chapter, you've seen concurrency used in Camel routes by the various EIPs that support them. The remainder of the chapter focuses on how scalability works with Camel and how you can build custom components that would support scaling.

13.4 Using the asynchronous routing engine

Camel uses its routing engine to route messages either synchronously or asynchronously. This section focuses on scalability. You'll learn that higher scalability can be achieved with the help of the asynchronous routing engine.

For a system, *scalability* is the desirable property of being capable of handling a growing amount of work gracefully. In section 13.1, we covered the Rider Auto Parts inventory application, and you saw that you could increase throughput using concurrent processing. In that sense, the application was scalable, because it could handle a growing amount of work in a graceful manner. That application could scale because it had a mix of CPU-bound and I/O-bound processes, and because it could use thread pools to distribute work.

In the next section, we'll look at scalability from a different angle. We'll see what happens when messages are processed asynchronously.

13.4.1 HITTING THE SCALABILITY LIMIT

Rider Auto Parts uses a Camel application to service its web store, as illustrated in figure 13.9. As you can see, Jetty consumer handles all requests from the customers. There are a variety of requests to handle, such as updating shopping carts, performing searches, gathering production information, and so on—the usual functions you expect from a web store. But one function involves calculating pricing information for customers. The pricing model is complex and individual to each customer; only

the ERP system can calculate the pricing. As a result, the Camel application communicates with the ERP system to gather the prices. While the prices are calculated by the ERP system, the web store has to wait until the reply comes back before it returns its response to the customer.

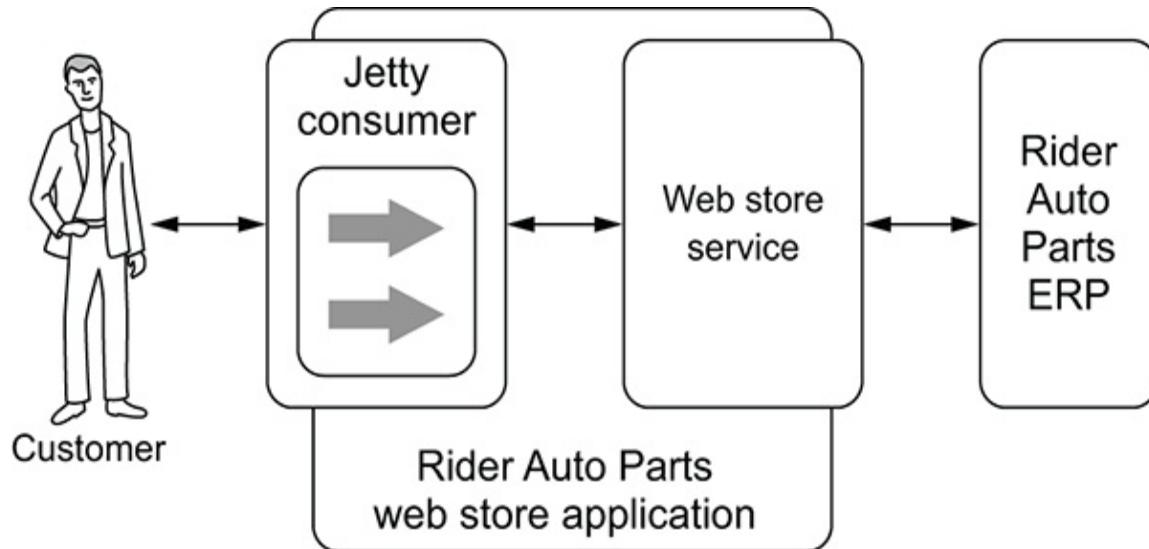


Figure 13.9 The Rider Auto Parts web store communicates with the ERP system to gather pricing information.

The business is doing well for the company, and an increasing number of customers are using the web store, which puts more load on the system. Lately, problems have been occurring during peak hours, with customers reporting that they can't access the web store or that it's generally responding slowly.

The root cause has been identified: the communication with the ERP system is fully synchronous, and the ERP system takes an average of 5 seconds to compute the pricing. This puts a burden on the Jetty thread pool, as there are fewer free threads to service new requests.

Figure 13.10 illustrates this problem. You can see that the thread is blocked (the white boxes) while waiting for the ERP system to return a reply.

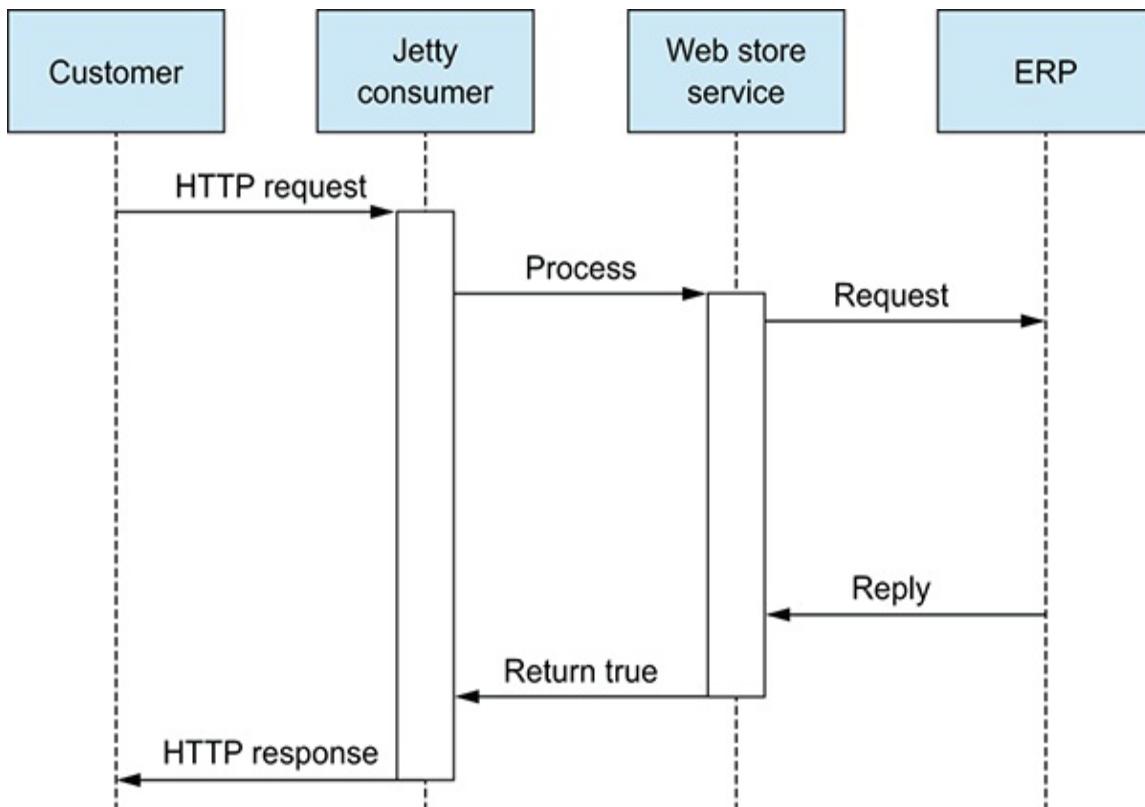


Figure 13.10 A scalability problem illustrated by the thread being blocked (represented as white boxes) while waiting for the ERP system to return the reply

Figure 13.10 reveals that the Jetty consumer is using one thread per request. This leads to a situation where you run out of threads as traffic increases. You've hit a scalability limit. Let's look into why and check out what Camel has under the hood to help mitigate such problems.

13.4.2 SCALABILITY IN CAMEL

It would be much better if the Jetty consumer could somehow *borrow* the thread while it waits for the ERP system to return the reply, and use the thread in the meantime to service new requests. This can be done using an asynchronous processing model. Figure 13.11 shows the principle.

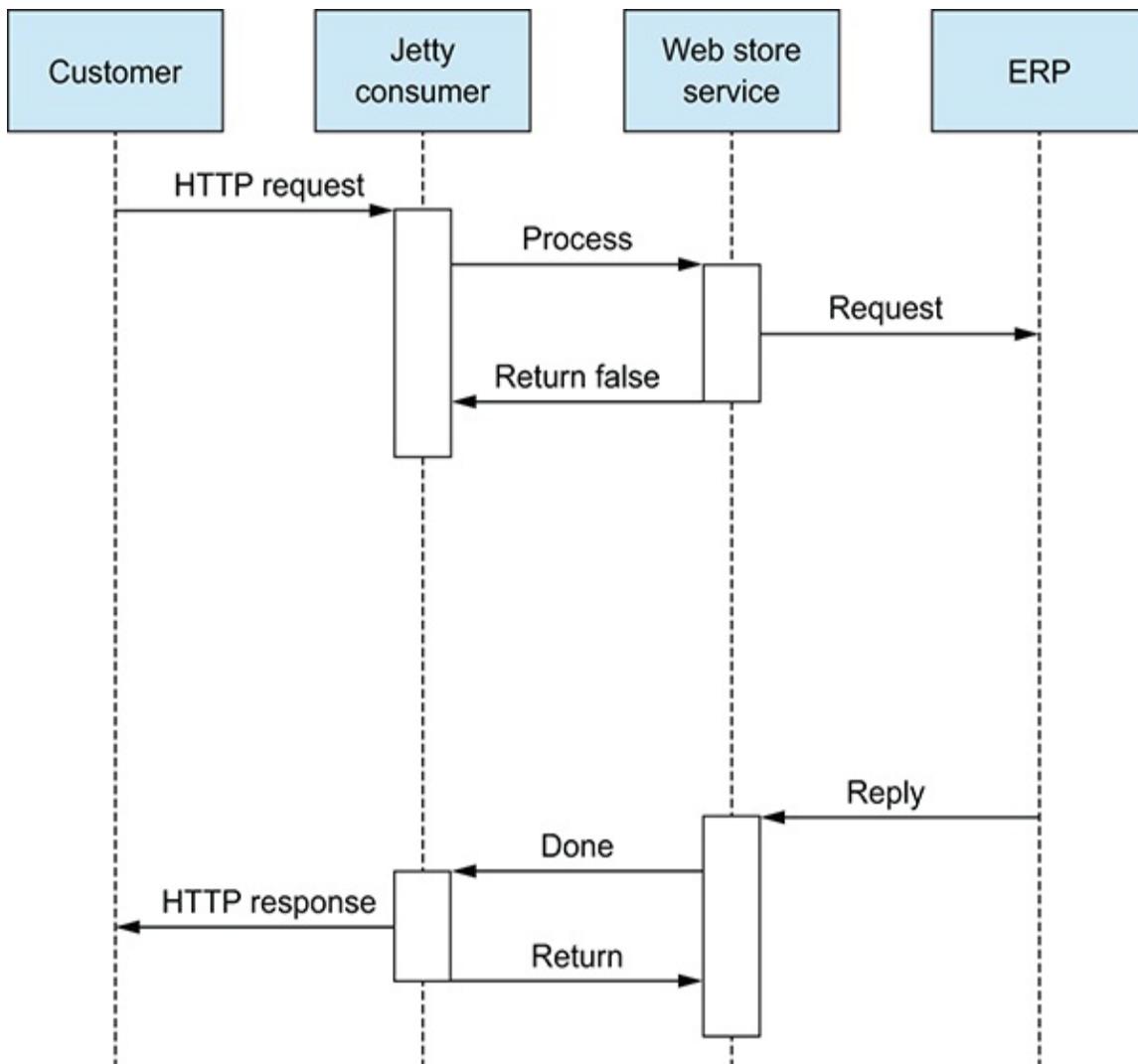


Figure 13.11 The scalability problem is greatly improved. Threads are much less blocked (represented by white boxes) when you use asynchronous communication between the systems.

If you compare figures 13.10 and 13.11, you can see that the threads are much less blocked in the latter (the white boxes are smaller). No threads are blocked while the ERP system is processing the request. This is a huge scalability improvement because the system is much less affected by the processing speed of the ERP system. If it takes 1, 2, 5, or 30 seconds to reply, it doesn't affect the web store's resource use as much as it would otherwise. The threads in the web store are much less I/O bound and are put to better use doing actual work.

Figure 13.12 shows a situation in which two customer requests

are served by the same thread without impacting response times. In this situation, customer 1 sends a request that requires a price calculation, so the ERP system is invoked asynchronously. A short while after, customer 2 sends a request that can be serviced directly by the web shop service, so it doesn't use the asynchronous processing model (it's synchronous). The response is sent directly back to customer 2. Later, the ERP system returns the reply, which is sent back to the waiting customer 1.

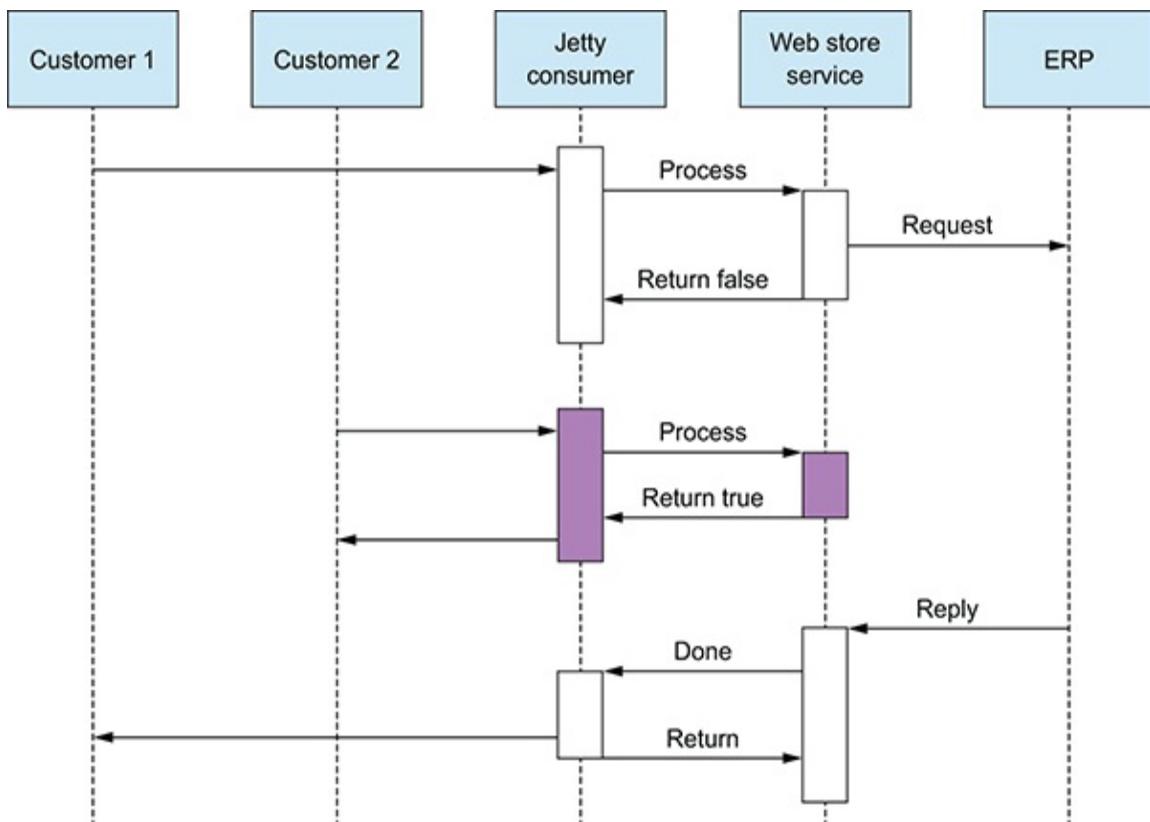


Figure 13.12 The same thread services multiple customers without blocking (white and gray boxes) and without impacting response times, resulting in much higher scalability.

In this example, you can successfully process two customers without increasing their response times. You've achieved higher scalability.

The next section peeks under the hood to see how this is possible in Camel when using the asynchronous processing model.

13.4.3 COMPONENTS SUPPORTING ASYNCHRONOUS PROCESSING

The routing engine in Camel is capable of routing messages either synchronously or asynchronously. The latter requires the Camel component to support asynchronous processing, which in turn depends on the underlying transport supporting asynchronous communication. To achieve high scalability in the Rider Auto Parts web store, you need to use asynchronous routing at two points. The communication with the ERP system and with the Jetty consumer must both happen asynchronously. The Jetty component already supports this.

Communication with the ERP system must happen asynchronously too. To understand how this is possible with Camel, we'll take a closer look at [figure 13.12](#). The figure reveals that after the request has been submitted to the ERP system, the thread won't block but will return to the Jetty consumer. It's then up to the ERP transport to notify Camel when the reply is ready. When Camel is notified, it'll be able to continue routing and let the Jetty consumer return the HTTP response to the waiting customer.

To enable all this to work together, Camel provides an asynchronous API that the components must use. The next section walks through this API.

13.4.4 ASYNCHRONOUS API

Camel supports an asynchronous processing model, which we refer to as the *asynchronous routing engine*. Using asynchronous processing has advantages and disadvantages compared to using the standard synchronous processing model. They're listed in table [13.5](#).

Table 13.5 Advantages and disadvantages of using the asynchronous processing model

Advantage	Disadvantage
Processing messages asynchronously doesn't use up threads, forcing them to wait for processors to	Implementing asynchronous processing

complete on blocking calls.

is much more complex.

It increases the scalability of the system by reducing the number of threads needed to manage the same workload.

The asynchronous processing model is manifested by an API that must be implemented to use asynchronous processing. You've already seen a glimpse of this API in [figure 13.12](#); the arrow between the Jetty consumer and the web store service has the labels *Return false* and *Done*. Let's see the connection that those labels have with the asynchronous API.

ASYNCPROCESSOR

The `AsyncProcessor` is an extension of the synchronous Processor API:

```
public interface AsyncProcessor extends Processor {  
    boolean process(Exchange exchange, AsyncCallback  
callback);  
}
```

`AsyncProcessor` defines a single `process` method that's similar to its synchronous Processor `.process` sibling.

Here are the rules that apply when using `AsyncProcessor`:

- A non-null `AsyncCallback` must be supplied; it'll be notified when the exchange processing is completed.
- The `process` method must not throw any exceptions that occur while processing the exchange. Any such exceptions must be stored on the exchange via the `setException` method.
- The `process` method must know whether it'll complete the processing synchronously or asynchronously. The method will return `true` if it completes synchronously; otherwise, it returns `false`.
- When the processor has completed processing the exchange, it must call the `callback.done(boolean doneSync)` method. The

`doneSync` parameter must match the value returned by the `process` method.

The preceding rules may seem a bit confusing at first. Don't worry—the asynchronous API isn't targeted at Camel end users but at Camel component writers. The next section covers an example of implementing a custom component that acts asynchronously. You'll be able to use this example as a reference if you need to implement a custom component.

NOTE You can read more about the asynchronous processing model at the Camel website:
<http://camel.apache.org/asynchronous-processing.html>.

The `AsyncCallback` API is a simple interface with one method:

```
public interface AsyncCallback {  
    void done(boolean doneSync);  
}
```

It's this callback that's invoked when the ERP system returns the reply. This notifies the asynchronous routing engine in Camel that the exchange is ready to be continued, and the engine can then continue routing it.

Let's see how this all fits together by digging into the example and looking at some source code.

13.4.5 WRITING A CUSTOM ASYNCHRONOUS COMPONENT

The book's source code contains the web store example in the `chapter13/scalability` directory. This example contains a custom ERP component that simulates asynchronous communication with an ERP system. The following listing shows how the `ErpProducer` is implemented.

Listing 13.7 `ErpProducer` using the asynchronous processing model

```
public class ErpProducer extends DefaultAsyncProducer  
{
```

1

Extends DefaultAsyncProducer

```
    private ExecutorService executor;  
  
    public ErpProducer(Endpoint endpoint) {  
        super(endpoint);  
    }  
  
    protected void doStart() throws Exception {  
        super.doStart();  
        this.executor =  
getEndpoint().getCamelContext()
```

2

Thread pool used to simulate asynchronous tasks

```
.getExecutorServiceManager().newFixedThreadPool(this,  
"ERP", 10);  
}  
  
protected void doStop() throws Exception {  
    super.doStop();  
    getEndpoint().getCamelContext()  
  
.getExecutorServiceManager().shutdown(executor);  
}  
  
public boolean process(final Exchange exchange,
```

3

Implements asynchronous process method

```
                final AsyncCallback  
callback) {  
    executor.submit(new ERPTask(exchange, callback));  
    log.info("Returning false");  
    return false;
```

4

4

Returns false to use asynchronous processing

```
}
```

```
private class ERPTask implements Runnable {
```

```
    private final Exchange exchange;
```

```
    private final AsyncCallback callback;
```

```
    private ERPTask(Exchange exchange, AsyncCallback
```

```
callback) {
```

```
        this.exchange = exchange;
```

```
        this.callback = callback;
```

```
}
```

```
    public void run() {
```

```
        log.info("Calling ERP");
```

```
        try {
```

```
            Thread.sleep(5000);
```

```
            log.info("ERP reply received");
```

```
            String in =
```

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

```
            exchange.getOut().setBody(in +
```

```
";516");
```

5

5

Sets reply on exchange

```
} catch (Throwable e) {
```

```
    exchange.setException(e);
```

6

6

Catches all exceptions on the exchange

```
} finally {
```

```
    log.info("Continue routing");
```

```
    callback.done(false);
```

7

7

Notifies callback reply is ready

```
        }
    }
}
```

When implementing a custom asynchronous component, it's most often the Producer that uses asynchronous communication, and a good starting point is to extend the `DefaultAsyncProducer` **1**.

To simulate asynchronous communication, you use a thread pool to execute tasks asynchronously **2**; you need to create a thread pool in the `doStart` method of the producer. To support the asynchronous processing model, `ErpProducer` must also implement the asynchronous process method **3**.

To simulate the communication, which takes 5 seconds to reply, you submit `ERPTask` to the thread pool. When the 5 seconds are up, the reply is ready, and it's set on the exchange **5**. Notice how we use a `try ... catch ... finally` block to ensure that any errors are caught and set on the exchange **6**, and the `finally` block ensures that the callback is called **7**. This is a recommended design to use when building custom asynchronous Camel components.

According to the rules, when you're using `AsyncProcessor`, the callback must be notified when you're done with a matching synchronous parameter **7**. In this example, `false` is used as the synchronous parameter because the `process` method returned `false` **4**. By returning `false`, you instruct the Camel routing engine to use asynchronous routing from this point forward for the given exchange.

You can try this example by running the following Maven goal from the `chapter13/scalability` directory:

```
mvn test -Dtest=ScalabilityTest
```

This runs two test methods: one request is processed fully synchronously (not using the ERP component), and the other is processed asynchronously (by invoking the ERP component).

When running the test, pay attention to the console output. The synchronous test will log input and output as follows:

```
2017-07-17 11:41:42 [      qtp1444378545-11]
INFO  input
- Exchange[ExchangePattern:InOut, Body:1234;4;1719;bumper]
2017-07-17 11:41:42 [      qtp1444378545-11]
INFO  output
- Exchange[ExchangePattern:InOut, Body:Some other action
here]
```

Notice that both the input and output are being processed by the same thread.

The asynchronous example is different, as the console output reveals:

```
2017-07-17 11:49:48 [      qtp515060127-11]
INFO  input
- Exchange[ExchangePattern:InOut, Body:1234;4;1719;bumper]
2017-07-17 11:49:48 [      qtp515060127-11]
INFO  ErpProducer
- Returning false (processing will continue
asynchronously)
2017-07-17 11:49:48 [Camel (camel-1) thread #0 - ERP] INFO
ErpProducer
- Calling ERP
2017-07-17 11:49:53 [Camel (camel-1) thread #0 - ERP] INFO
ErpProducer
- ERP reply received
2017-07-17 11:49:53 [Camel (camel-1) thread #0 - ERP] INFO
ErpProducer
- Continue routing
2017-07-17 11:49:53 [Camel (camel-1) thread #0 - ERP] INFO
output
- Exchange[ExchangePattern:InOut,
Body:1234;4;1719;bumper;516]
```

This time two threads are used during the routing. The first is the thread from Jetty, which received the HTTP request. As you can see, this thread was used to route the message to ErpProducer. The other thread takes over communication with the ERP system. When the reply is received from the ERP system, the callback is notified, which lets Camel hijack the thread and use it to continue routing the exchange. You can see

this from the last line, which shows the exchange routed to the log component.

Now, this was the happy path. When writing or using an asynchronous component, there's a price to pay, which is what we'll discuss next.

13.4.6 POTENTIAL ISSUES WHEN USING AN ASYNCHRONOUS COMPONENT

You may have seen from implementing a custom asynchronous component (as shown in [listing 13.7](#)) that implementing this correctly is much more complicated. In particular, you need to understand the importance of making sure to call the `done` method on the `AsyncResult` parameter. Calling the `done` method is what triggers the asynchronous routing engine in Camel to continue routing the exchange. Several potential issues can cause the `done` method to never be called, such as these:

- A bug in the code causes the `done` method to never be called.
- The remote system never returns with a reply, causing the task to not finish.
- The `done` method is called but with the wrong value (should use `false`).

This concludes our coverage of scalability and the ups and downs of implementing custom asynchronous components with Camel.

13.5 Summary and best practices

In this chapter, we looked at thread pools, the foundation for concurrency in Java and Camel. You saw how concurrency greatly improves performance, and we considered all the possible ways to create, define, and use thread pools in Camel. You saw how easy it is to use concurrency with the numerous EIPs in Camel. You witnessed how Camel can scale up using asynchronous (nonblocking) routing. This comes with a cost of complexity, which we covered in depth so you could learn about

the pitfalls and best practices for implementing your custom Camel components.

Here are some best practices related to concurrency and scalability:

- *Use concurrency if possible*—Concurrency can greatly speed up your applications. Note that using concurrency requires business logic that can be invoked in a concurrent manner.
- *Tweak thread pools judiciously*—Tweak thread pools only when you have a means of measuring the changes. It's often better to rely on the default settings.
- *Use asynchronous processing for high scalability*—If you require high scalability, try using the Camel components that support the asynchronous processing model.
- *Take care when implementing your own asynchronous component*—You're required to structure your component code according to numerous rules. This ensures that Camel can route the messages when data has been received or a time-out has occurred, and avoids any potential issues with threads getting stuck.
- *Reactive systems*—Reactive streams and frameworks are gaining in popularity. You can find details on using these with Camel in chapter 20 (available online).

Now, prepare for something completely different, as you're about to embark on a journey focusing on securing your applications with Camel.

14

Securing Camel

This chapter covers

- Securing your Camel configuration
- Web service security
- Transport security
- Encryption and decryption
- Signing messages
- Authentication and authorization

Security in enterprise applications seems to become more and more important every year. As mobile and web access endpoints are the preferred method of access for customers, applications are becoming more open to the greater internet and consequently more open to attack. Unauthorized access to these exposed endpoints can become a costly thing to deal with. For example, having private customer data leaked on the internet has plagued retailers in recent years. These events definitely have an impact on the current and future bottom line of a company's finances.

With that said, it's important to note that Camel is by default *not* secured! There's a good reason for this: application security has many angles, and not all may be applicable to every use case. For instance, you probably don't need to encrypt your payload if

the communication link is within your company's VPN. But authentication and authorization may be needed. Camel can help you implement as much or as little security as you require, with relative ease. We say *relative* because security configuration can become quite complex just by its nature.

This chapter covers the main areas of security in Camel. We'll start by covering how to secure sensitive information in your configuration. Next we'll cover security in web services. Then we'll show how to sign and encrypt messages within Camel. We'll then cover transport security, which is all about securing the link between Camel and the remote service using Transport Layer Security (TLS). Finally, Camel also provides authentication and authorization services for controlling access to your routes.

Let's start by looking at how to secure your configuration.

14.1 Securing your configuration

As you saw in chapter 2, externalizing your configuration and referencing with property placeholders are great ways to ease the transition from testing to production. Often you need to store such things as usernames, access keys, passwords, or other sensitive data to access remote services in your configuration. If there's a chance that someone may view this information, having everything in plain text would give that person access to the remote resource. To prevent that from happening, the camel-jasypt component allows you to encrypt your configuration values and then decrypt them via the property placeholder mechanism at runtime.

Let's first look at using camel-jasypt to encrypt your sensitive data.

14.1.1 ENCRYPTING CONFIGURATION

Encrypting your configuration is a task that typically happens outside your runtime route. The next section covers the runtime scenario (decrypting configuration). The camel-jasypt

component includes a command-line utility to assist with this encryption, although, in theory, any utility that uses the same Java Cryptography Architecture (JCA) algorithm would do. The camel-jasypt utility is included with the camel-jasypt JAR in the lib folder of the Apache Camel distribution. For example, you can get help with this utility as follows:

```
[janstey@bender]$ cd apache-camel-2.20.1/
[janstey@bender]$ wget
http://repo1.maven.org/maven2/org/jasypt/jasypt/1.9.2/jasypt-1.9.2.jar
[janstey@bender]$ java -cp jasypt-1.9.2.jar:lib/camel-jasypt-2.20.1.jar org.apache.camel.component.jasypt.Main -help
Apache Camel Jasypt takes the following options

-h or -help = Displays the help screen
-c or -command <command> = Command either encrypt or
decrypt
-p or -password <password> = Password to use
-i or -input <input> = Text to encrypt or decrypt
-a or -algorithm <algorithm> = Optional algorithm to use
```

As you can see, the utility supports both encryption and decryption for convenience. To encrypt the secret text secret, you'd use the following options:

```
[janstey@bender]$ java -cp jasypt-1.9.2.jar:lib/camel-jasypt-2.20.1.jar org.apache.camel.component.jasypt.Main -c
encrypt -p supersecret -i secret
Encrypted text: q+XT/4rR94ghCbNp5coaxg==
```

Pay special attention to the -p option; we used an encryption password of supersecret, which you'll need to use later when decrypting configuration. The output from the utility is the encrypted value of the secret text. You can just as easily go back to the original value:

```
[janstey@bender]$ java -cp jasypt-1.9.2.jar:lib/camel-jasypt-2.20.1.jar org.apache.camel.component.jasypt.Main -c
decrypt -p supersecret -i "q+XT/4rR94ghCbNp5coaxg=="
Decrypted text: secret
```

In both cases, you didn't specify anything for the algorithm (-a)

option, so the default algorithm of `PBEWithMD5AndDES` will be used. This is a standard algorithm name as defined in the JCA Standard Algorithm Name Documentation (<http://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html>). If you decide to use another algorithm here, make sure to use the same when you're decrypting configuration within your route. Also, not all security providers provide the same set of algorithms, so before configuring something different, ensure that all systems in your deployment can support the algorithm.

Now that you have your password encrypted, let's look at how to use that within your Camel route.

14.1.2 DECRYPTING CONFIGURATION

To use the `camel-jasypt` component within your Camel project, you need to add a dependency to it in your Maven POM:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jasypt</artifactId>
    <version>2.20.1</version>
</dependency>
```

You then need to configure the Camel Properties component to use the `PropertyParser` provided by `camel-jasypt`. In XML DSL, you reference it with the `propertiesParserRef` attribute on `<propertyPlaceholder>` within the `camelContext`, as follows:

```
<bean id="jasypt"
    class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <property name="password" value="supersecret"/>
</bean>

<camelContext
    xmlns="http://camel.apache.org/schema/spring">
    <propertyPlaceholder id="properties"
        location="classpath:rider-
test.properties"
        propertiesParserRef="jasypt"/>
```

```
...
```

In the `rider-test.properties` file, you can then define the encrypted properties using the special `ENC()` notation:

```
ftp.password=ENC(q+XT/4rR94ghCbNp5coaxg==)
```

The text inside the parentheses is the secret text that you encrypted in the previous section. You can now use the encrypted property directly in the endpoint URI as the FTP server password, as shown in bold in this route:

```
<camelContext
  xmlns="http://camel.apache.org/schema/spring">
  ...
  <route>
    <from uri="file:target/inbox"/>
    <to uri="ftp://rider:
{{ftp.password}@localhost:21000/target/outbox"/>
  </route>
</camelContext>
```

You can do all of this from the Java DSL as well. The following listing shows how.

Listing 14.1 Referencing encrypting properties from the Java DSL

```
public class SecuringConfigTest extends CamelTestSupport {
    @EndpointInject(uri = "file:target/inbox")
    private ProducerTemplate inbox;
    private FtpServerBean ftp = new FtpServerBean();

    @Override
    protected CamelContext createCamelContext() throws
Exception {
    CamelContext context = super.createCamelContext();

    JasyptPropertiesParser jasypt = new
JasyptPropertiesParser(); ①
```

①

Creates the jasypt properties parser

```
jasypt.setPassword("supersecret"); ②
```

②

Sets the encryption password

```
PropertiesComponent prop =  
    context.getComponent("properties",  
PropertiesComponent.class);  
    prop.setLocation("classpath:rider-  
test.properties");  
    prop.setPropertiesParser(jasypt); ③
```

③

Uses the jasypt properties parser so you can decrypt values

```
        return context;  
    }  
  
    ...  
  
    @Override  
    protected RouteBuilder createRouteBuilder() throws  
Exception {  
    return new RouteBuilder() {  
        @Override  
        public void configure() throws Exception {  
            from("file:target/inbox")  
                .to("ftp://rider:  
{{ftp.password}}@localhost:21000"
```

References the encrypted property with an endpoint URI

```
                + "/target/outbox");  
        }  
    };  
}
```

After creating CamelContext, you create the jasypt PropertyParser **①** and set it on PropertiesComponent **③**. You also need to use the

encryption password ② as you did in the XML DSL example before.

You can try this example using the following Maven goals from the chapter14/configuration directory:

```
mvn test -Dtest=SecuringConfigTest  
mvn test -Dtest=SpringSecuringConfigTest
```

Because you have your security hat on right now, you may have been thinking that this plain-text encryption password is a security hole. You'd be right, of course! After all, what's the point of encrypting your remote service password if you then leave the encryption password in plain view?

EXTERNALIZING THE ENCRYPTION PASSWORD

How do you deal with a plain-text encryption password? The solution is to not define it in your application but to refer to it from an OS environment variable or JVM system property at runtime. For example, to use an environment variable

CAMEL_ENCRYPTION_PASSWORD instead of the supersecret password in [Listing 14.1](#), you could do as follows:

```
jasypt.setPassword("sysenv:CAMEL_ENCRYPTION_PASSWORD");
```

The sysenv prefix directs camel-jasypt to use the value in the CAMEL_ENCRYPTION_PASSWORD environment variable. Now, before starting your Camel application, you'd set an additional environment variable:

```
export CAMEL_ENCRYPTION_PASSWORD=supersecret
```

You can also refer to JVM system properties in the same manner, but instead you need to use the sys prefix.

Now that you've secured your configuration, let's look at how to secure web services.

14.2 Web service security

Web services are an extremely useful integration technology for distributed applications. They are often associated with service-oriented architecture (SOA), in which each service is defined as a web service.

You can think of a web service as an API on the network. The API itself is defined using the Web Services Description Language (WSDL), specifying the operations you can call on a web service and the input and output types, among other things. Messages are typically XML, formatted to comply with the SOAP schema. In addition, these messages are typically sent over HTTP. Web services allow you to write Java code and make that Java code callable over the internet, which is pretty neat!

For accessing and publishing web services, Camel uses Apache CXF (<http://cxf.apache.org>). CXF is a popular web services framework that supports many web services standards. We covered Camel's use of CXF in chapter 10, so you can refer to that chapter for more general usage. Here we focus on CXF's support of the Web Services Security (WS-Security) standard. The WS-Security support in CXF allows you to do the following:

- Use authentication tokens
- Encrypt messages
- Sign messages
- Timestamp messages

To show these concepts in action, let's go back to Rider Auto Parts, which needs a new piece of functionality implemented. In chapter 2, you saw that customers could place orders in two ways:

- Uploading the order file to an FTP server
- Submitting the order from the Rider Auto Parts web store via HTTP

What we didn't say then was that this HTTP link to the back-end order processing systems needs to be a web service. Of course,

you wouldn't want anonymous users to be able to submit orders, so you're going to use WS-Security to help you out.

14.2.1 AUTHENTICATION IN WEB SERVICES

Clients of the new Rider Auto Parts web service have a simple way to submit orders. The service endpoint in question looks like this:

```
import javax.jws.WebService;

@WebService
public interface OrderEndpoint {
    OrderResult order(Order order);
}
```

Clients can call the `order` method with an `Order` object as the only argument. The `Order` object contains a part name, amount to order, and the customer name. An `OrderResult` is returned and can tell the user whether the order succeeded or failed. The big missing piece here is that it's by default unsecured! Anyone who knows the endpoint address can submit an order—hardly ideal for Rider Auto Parts or the customer who will be charged for the parts. The Rider Auto Parts developer team clearly needs to add authentication to this service.

With authentication, a user proves who they are to a system, typically using a username and password. In the web-services world, this is encapsulated in a `<UsernameToken>` element within the SOAP message. The receiver of a SOAP message with a `UsernameToken` can check things such as whether a username exists, whether a password is valid, or if the timestamp on the token is still valid. Consider this example `UsernameToken`:

```
<wsse:UsernameToken
    wsu:Id="UsernameToken-db49b9e2-1051-4559-bd48-
39877be634ad">
    <wsse:Username>jon</wsse:Username>
    <wsse:Password
        Type="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-username
        -token-profile-
        1.0#PasswordDigest">ZbNxe8WcmHekEcR6pbQLaVzW6zk=
```

```
</wsse:Password>
<wsse:Nonce
    EncodingType="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-
        soap-message-security-
1.0#Base64Binary">Zi8UQpTp28/0eSNqIO
    9eTQ==
</wsse:Nonce>
<wsu:Created>2017-10-25T18:06:46.052Z</wsu:Created>
</wsse:UsernameToken>
```

That's not something you want to write or interpret by hand. Fortunately, CXF provides interceptors that can be used on the client and server side to process this information before the underlying service endpoint is called.

SERVER-SIDE WS-SECURITY PROCESSING

First, let's look at how the server-side configuration will change when security is added. The route used to implement the web service won't change at all:

```
<route>
    <from uri="cxft:bean:orderEndpoint"/>
    <to uri="seda:incomingOrders"/>
    <transform>
        <method beanType="camelaction.order.OrderResultBean"
            method="orderOK"/>
    </transform>
</route>
```

This simple route sends an order to an `incomingOrders` queue and then returns an `OK` result back to the client. At this point, the user is assumed authenticated. You need to look at the CXF configuration in the following listing to see how this authentication was set up.

[Listing 14.2](#) Adding `usernameToken` authentication to a CXF web service

```
<bean id="loggingOutInterceptor"
    class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>
```

```
<bean id="loggingInInterceptor"  
      class="org.apache.cxf.interceptor.LoggingInInterceptor"/>  
  
<bean id="wss4jInInterceptor"  
      class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor"  
>    ①
```

①

Creates a WSS4JInInterceptor to process the WS-Security SOAP header

```
<constructor-arg>  
  <map>  
    <entry key="action" value="UsernameToken  
Timestamp"/>  ②
```

②

Sets the actions to use when processing the UsernameToken

```
  <entry key="passwordCallbackClass"  
        value="camelinaction.wssecurity.ServerPasswordCallback"/>  
  ③
```

③

Specifies the callback to use when checking the username and password

```
  </map>  
  </constructor-arg>  
</bean>  
  
<cxft:cxfEndpoint id="orderEndpoint"  
                   address="http://localhost:9000/order"  
                   serviceClass="camelinaction.order.OrderEndpoint">  
  
  <cxft:inInterceptors>  
    <ref bean="loggingInInterceptor"/>  
    <ref bean="wss4jInInterceptor"/>
```

Adds the WSS4JInInterceptor to the cxfEndpoint of the OrderEndpoint

```
</cxf:inInterceptors>

<cxf:outInterceptors>
    <ref bean="loggingOutInterceptor"/>
</cxf:outInterceptors>
</cxf:cxfEndpoint>
```

Your cxfEndpoint bean again doesn't change much from the unsecured case. The security magic happens in CXF's org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor ❶. WSS4J in the class name refers to the Apache WSS4J project, which provides the implementation of many of the security concepts in CXF. The action entry ❷ specifies the tasks that will be executed using the interceptor. In our case, this will be to process a UsernameToken and a Timestamp. Take note of these actions, as the client code will also need to provide them. The passwordCallbackClass ❸ entry specifies a class that'll be used to check the username and provide a password. In our case, you'll be using a hardcoded username/password combination to demonstrate the flow. Such a callback handler class is shown in the following listing.

Listing 14.3 A simple WS-Security callback for a server-side app

```
package camelaction.wssecurity;

import java.io.IOException;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler; ❶
```

❶

You need to implement a CallbackHandler.

```
import
javax.security.auth.callback.UnsupportedCallbackException;
```

```
import org.apache.wss4j.common.ext.WSPasswordCallback;

public class ServerPasswordCallback implements
CallbackHandler {

    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        WSPasswordCallback pc = (WSPasswordCallback)
callbacks[0];
        if ("jon".equals(pc.getIdentifier())) { 2

```

2

Checks the username

```
        pc.setPassword("secret"); 3
```

3

Sets the password on the callback if the username is valid

```
    } else {
        throw new IOException("Username incorrect!");
    }
}
```

First, your `ServerPasswordCallback` implements `CallbackHandler` **1**, which has a single method: `handle`. In the `handle` method, you need to check whether the username is valid **2** and, if so, set the password on the callback **3**. The verification of the password is done by WSS4J under the covers. The password itself may not be stored in plain text, so comparing it may be an effort in itself. It's best to leave the checking up to WSS4J.

Now, the Rider Auto Parts order web service has full authentication support, albeit for a single user, `jon`, but still full authentication support. Let's see next how clients of this web service can add user credentials.

CLIENT-SIDE WS-SECURITY CONFIGURATION

Like the server-side case, the main thing you're doing to add WS-

Security authentication to your application is adding a WSS4J interceptor to your CXF-based web service. It's easiest to explain with code, so let's look at the following listing in detail.

Listing 14.4 WS-Security client-side configuration

```
protected static OrderEndpoint createCXFClient(
    String url, String user, String passwordCallbackClass)
{
    List<Interceptor<? extends Message>> outInterceptors =
new ArrayList<>();

    // Define WSS4j properties for flow outgoing
    Map<String, Object> outProps = new HashMap<>();
    outProps.put("action", "UsernameToken"
Timestamp"); 1
```

1

Sets the actions to use when processing the UsernameToken

```
outProps.put("user", user); 2
```

2

Specifies the user

```
outProps.put("passwordCallbackClass",
passwordCallbackClass); 3
```

3

Specifies the callback to use to grab the password

```
WSS4JOutInterceptor wss4j = new
WSS4JOutInterceptor(outProps);
// Add LoggingOutInterceptor
LoggingOutInterceptor loggingOutInterceptor = new
LoggingOutInterceptor();

outInterceptors.add(wss4j);
outInterceptors.add(loggingOutInterceptor);
```

```
// we use CXF to create a client for us as its easier  
than JAXWS and works  
JaxWsProxyFactoryBean factory = new  
JaxWsProxyFactoryBean(); ④
```

④

Uses a CXF factory bean to create the OrderEndpoint

```
factory.setOutInterceptors(outInterceptors);  
factory.setServiceClass(OrderEndpoint.class);  
factory.setAddress(url);  
return (OrderEndpoint) factory.create();  
}  
  
@Test  
public void testOrderOk() throws Exception {  
    OrderEndpoint client = createCXFClient(  
        "http://localhost:9000/order", "jon",  
        "camelinaction.wssecurity.ClientPasswordCallback");  
    OrderResult reply = client.order(new Order("motor",  
100, "honda")); ⑤
```

⑤

Places an order using the OrderEndpoint

```
assertEquals("OK", reply.getMessage());  
}
```

Before you create a business-as-usual JAX-WS web service using a CXF factory bean ④, you need to configure wss4jOutInterceptor. You use the same actions as on the server side ①. You want to specify UsernameToken and Timestamp so a passed authentication doesn't last indefinitely. You're on the client side now, so you need to specify the user you want accessing the remote service. You do this by adding a user entry for the user named jon ②. You specify a different callback handler on the client side ③, which we'll discuss next. Finally, you place an order using OrderEndpoint created by the CXF JAX-WS factory bean ⑤; this will invoke the remote web service and return a result. All the marshaling and networking is handled by CXF under the hood.

The callback handler for the client side is a little different in that you're providing a password to use when calling the web service. The following listing shows such a callback handler.

[Listing 14.5](#) A simple WS-Security callback for a client-side app

```
package camelinaction.wssecurity;

import java.io.IOException;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.wss4j.common.ext.WSPasswordCallback;

public class ClientPasswordCallback implements
CallbackHandler {

    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        WSPasswordCallback pc = (WSPasswordCallback)
callbacks[0];
        pc.setPassword("secret");
    }
}
```

Specifies the password to use when invoking the web service

This overly simplified callback handler gets the job done for your hardcoded example, but you wouldn't want to always return the same password for every user.

You can try this example for yourself by navigating to the chapter14/webservice directory and executing the following command:

```
mvn test -Dtest=WssAuthTest
```

In particular, note that the web service invocation will fail with

`javax.xml.ws.soap.SOAPFaultException` when either the username or password is incorrect. Recall that this is how you designed the server-side callback handler in [listing 14.3](#). Of course, with hardcoded usernames and passwords, using an incorrect one is highly unlikely. Let's look at how to integrate your web service with JAAS authentication.

14.2.2 AUTHENTICATING WEB SERVICES USING JAAS

The developers at Rider Auto Parts have successfully demonstrated their authenticated web service to management. Now they're tasked with authenticating against the users available in their companywide container standard: Apache Karaf. Users in Karaf can come from various sources such as LDAP, database, and properties files. All of these are accessed via the Java Authentication and Authorization Service (JAAS).

This process is made easy using an existing class from WSS4J. Recall that in [listing 14.3](#) the server-side `callbackHandler` set the password text on `WSPasswordCallback` without checking for its validity. We mentioned leaving the password validation up to WSS4J. Validating against JAAS is as easy as switching to using a `org.apache.wss4j.dom.validate.JAASUsernameTokenValidator` password validator. The following listing shows you how.

Listing 14.6 Using JAASUsernameTokenValidator to authenticate against JAAS

```
<bean id="karafJaasValidator" ①
```

①

Points the `JAASUsernameTokenValidator` to the Karaf container JAAS context

```
class="org.apache.wss4j.dom.validate.JAASUsernameTokenValidator"> ①
    <property name="contextName" value="karaf"/> ①
</bean>
```

```
<bean id="wss4jInInterceptor"
  class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
  <argument>
    <map>
      <entry key="action" value="UsernameToken"/> 2
    </map>
  </argument>
</bean>
```

2

The WSS4JInInterceptor should check for UsernameToken.

```
</map>
</argument>
</bean>

<cxft:cxftEndpoint id="orderEndpoint"
  address="/order" 4
```

4

Uses relative path for web service

```
serviceClass="camelinaction.order.OrderEndpoint">

  <cxft:inInterceptors>
    <ref component-id="loggingInInterceptor"/>
    <ref component-id="wss4jInInterceptor"/>
  </cxft:inInterceptors>
  <cxft:outInterceptors>
    <ref component-id="loggingOutInterceptor"/>
  </cxft:outInterceptors>

  <cxft:properties>
    <entry key="ws-security.ut.validator"> 3

```

3

**Overrides the default password validator with
JAASUsernameTokenValidator**

```
    <ref component-id="karafJaasValidator"/> 3
  </entry>
</cxft:properties>
```

```
</cxf:cxfEndpoint>
```

First, you construct an `org.apache.wss4j.dom.validate.JAASUsernameTokenValidator` that's configured to check `UsernameToken` credentials against `javax.security.auth.login.LoginContext` with the name `karaf` **1**. When deployed into Apache Karaf, this will pick up whatever login module it's using for authentication (for example, LDAP, database, or properties file). Now when you create `wss4jInInterceptor` **2**, it's different from before. The main difference is that you're now supplying no `callbackHandler`. This is a feature of Apache CXF; because you provided no `CallbackHandler` and your `UsernameToken` has a password set, CXF creates a simple `callbackHandler` for you. This simple `CallbackHandler` sets the password on `wscallbackHandler`, similar to what you were doing manually before. Finally, you need to override the default password validator with your JAAS-powered one. You can do that using the `ws-security.ut.validator` CXF configuration property **3**. With this new configuration, your web service is now authenticated against container credentials.

NOTE Web services deployed into a container such as Apache Karaf typically reuse an HTTP service that's already running. In Karaf's case, HTTP services are by default on port 8181, and CXF web services are registered under the /`cxf` path. For your web service in [listing 14.6](#), the /`order` path **4** will be available at <http://localhost:8181/cxf/order>.

To demonstrate this, you need to deploy the example into Apache Karaf. At the time of writing, you're using the latest version: Apache Karaf 4.1.2. You start Apache Karaf using the following command:

```
bin/karaf
```

Then install Camel version 2.20.1 and CXF WS-Security support

in Karaf:

```
feature:repo-add camel 2.20.1  
feature:install camel-cxf cxf-ws-security camel-blueprint
```

Now you need to install your Rider Auto Parts web service in Karaf, which you can do using the chapter14/webservice-karaf example from the book's source code. At first, you need to build this example:

```
mvn clean install
```

You also need to build the chapter14/webservice example from the previous section if you haven't already. Now from the Karaf command line, type this:

```
install -s mvn:com.camelinaction/chapter14-webservice/2.0.0  
install -s mvn:com.camelinaction/chapter14-webservice-  
karaf/2.0.0
```

This installs and starts the example (the `-s` flag refers to *start*).

The chapter14/webservice-karaf directory also contains a client that you can use to access the web service deployed in Karaf. This client is shown in the following listing.

Listing 14.7 A client to access the Rider Auto Parts web service

```
public class Client {  
  
    public static void main(String[] args) {  
        List<Interceptor<? extends Message>> outInterceptors  
= new ArrayList<>();  
  
        // Define WSS4j properties for flow outgoing  
        Map<String, Object> outProps = new HashMap<>();  
        outProps.put("action", "UsernameToken");  
        outProps.put("user", "karaf"); ❶
```

❶

Uses the default Karaf user

```
outProps.put("passwordType", "PasswordText");
```

②

②

Uses text-based password to be compatible with the JAAS validator

```
outProps.put("passwordCallbackClass",
```

③

③

Uses a CallbackHandler that accepts a password from the console

```
"camelinaction.StdInPasswordCallback");
```

③

```
WSS4JOutInterceptor wss4j = new
WSS4JOutInterceptor(outProps);
    outInterceptors.add(wss4j);

    JaxWsProxyFactoryBean factory = new
JaxWsProxyFactoryBean();
        factory.setOutInterceptors(outInterceptors);
        factory.setServiceClass(OrderEndpoint.class);

factory.setAddress("http://localhost:8181/cxf/order");

    OrderEndpoint client = (OrderEndpoint)
factory.create();

    Order order = new Order("motor", 100, "honda");
    System.out.println("Placing order for: " + order);
    OrderResult reply = client.order(order);

    System.out.println("Rider Auto Web service
returned: "
                    + reply.getMessage());
    }
}
```

Your simple client class looks similar to the web services test from the previous section. The first main difference is that you're specifying the default Karaf admin user karaf ① as the user. You then set the password type as text so that the WSS4J JAAS validator can compare it correctly ②. Finally, you specify CallbackHandler, which reads the password from the console ③. This simple CallbackHandler is shown in the following listing.

Listing 14.8 A callbackHandler that reads passwords from the console

```
public class StdInPasswordCallback implements  
CallbackHandler {  
  
    public void handle(Callback[] callbacks)  
        throws IOException, UnsupportedCallbackException {  
        WSPasswordCallback pc = (WSPasswordCallback)  
callbacks[0];  
        Console console = System.console();  
        console.printf("Please enter your Rider Auto Parts  
password: ");  
        char[] passwordChars = console.readPassword();  
        String passwordString = new String(passwordChars);  
        pc.setPassword(passwordString);  
    }  
}
```

To try this client, you can use the following command:

```
mvn exec:java -Pclient
```

You'll be prompted for a password to use:

```
Please enter your Rider Auto Parts password:
```

The default username and password in Karaf is karaf/karaf, so in the prompt you need to enter karaf as the password. If you don't enter the correct password,

`javax.xml.ws.soap.SOAPFaultException` will be thrown from the CXF client.

The WS-Security support in CXF also allows you to do things such as sign or encrypt your messages, among other things. For more information on how to configure these, refer to the Apache CXF website: <http://cxf.apache.org/docs/ws-security.html>.

Now that you've secured your Rider Auto Parts web service, let's look at securing more general payloads you could be sending with Camel.

14.3 Payload security

You have two main options for securing the message payloads you’re sending with Camel. One involves encrypting the entire contents of the message—a useful option when you’re sending sensitive information that you wouldn’t want viewed by any listening parties. But most often, you can pass the buck of encrypting your data down to the transport layer via the TLS protocol rather than handling it within a Camel route. These will probably be the most common secured endpoints you’ll encounter out in the wild. To see more about configuring TLS, skip to section 14.4. There’s a difference to handling encryption within Camel or delegating to the transport layer. Figures 14.1 and 14.2 illustrate this difference.

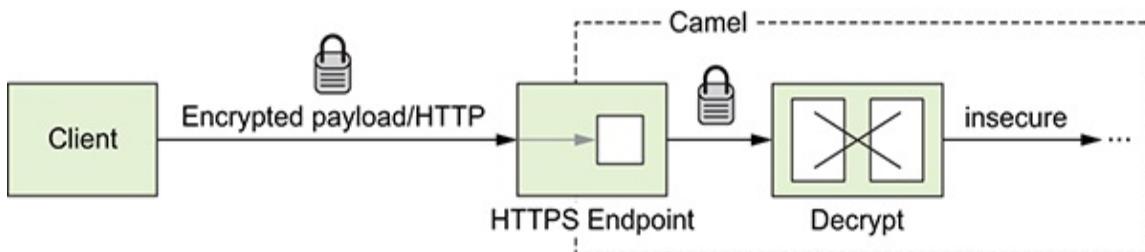


Figure 14.1 Using the camel-crypto data format, you can handle encryption and decryption of payloads within your Camel route. This gives you full control over what transport you send over—for example, an unsecured HTTP pipe.

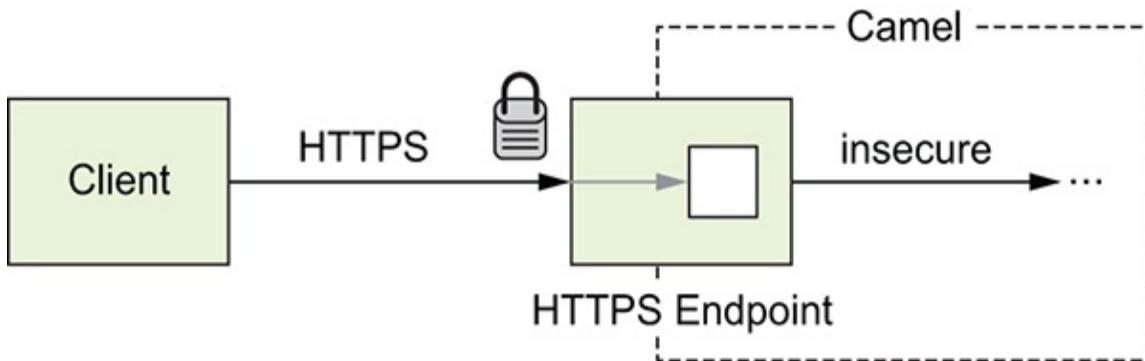


Figure 14.2 Using transport-based security to handle payload encryption gives you less choice over what transport you can use, because the underlying transport has to support JSSE. For a complete discussion on transport security, see section 14.4.

Another option in Camel to secure your payload is to send a digital signature along with the message so that the receiver can

verify that the message is unchanged and that the sender is correct. Let's look at how to sign and verify a message first.

14.3.1 DIGITAL SIGNATURES

Adding a digital signature to a message gives the receiver the ability to confirm that you sent the message and that no one tampered with it in transit. This is especially crucial in messages such as financial transactions, where a change of message origin or modification of a decimal place, say, would have large consequences. Digital signatures are generated using asymmetric cryptography: the sender generates the signature using a private key that only they have. The receiver then uses a different publicly available key to verify that the message is indeed the same. The mathematics behind this relationship is complex, but fortunately you don't have to delve into that to use the algorithms.

For signing messages in Camel, there's the camel-crypto component. To use the camel-crypto component within your Camel project, you need to add a dependency to it in your Maven POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-crypto</artifactId>
  <version>2.20.1</version>
</dependency>
```

Before digging into what the camel-crypto component gives you, you first have to create the public and private keys that will feed into the cryptography algorithms. For this, the keytool utility that ships with the Java Development Kit (JDK) is required.

GENERATING AND LOADING PUBLIC AND PRIVATE KEYS

Going over all the options of the keytool utility is beyond the scope of this book. To not confuse you (and even us), we'll stick to using default cryptographic algorithms where possible. If you do decide that the default algorithm isn't sufficient for your

application, make sure to specify the same algorithm in keytool as you do in Camel.

The first step in using keytool is to generate a *key pair*—a private and public key for the sender. This is accomplished with the `-genkeypair` command:

```
keytool -genkeypair \
    -alias jon \
    -dname "CN=Jon Anstey, L=Paradise, ST=Newfoundland,
C=Canada" \
    -validity 7300 \
    -keypass secret \
    -keystore cia_keystore.jks \
    -storepass supersecret
```

Here you’re creating a public/private key pair with the alias `jon`, expiration in 7,300 days, and key password of `secret`. The `dname` argument specifies the distinguished name of the certificate issuer. Finally, you specify `cia_keystore.jks` as the keystore file with a password of `supersecret`. If you use the `-list` keytool command, you can see that there’s one key-pair in your keystore:

```
[janstey@bender]$ keytool -list -keystore cia_keystore.jks \
\
-storespass supersecret

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

jon, Jun 29, 2017, PrivateKeyEntry,
Certificate fingerprint (SHA1):
EE:B0:BC:F3:B0:8E:24:C4:7B:0E:60:47:11:71:EA:76:DC:49:D8:83
```

Now this keystore is all ready for you to start signing messages. But you need to extract the public key into a separate keystore, because you can’t give out your private key. This keystore that contains only public keys is called the *trust store*. To do this, you can use the `-exportcert` and `-importcert` keytool commands:

```
[janstey@bender]$ keytool -exportcert -rfc -alias jon -file
jon.cer \
```

```

        -keystore cia_keystore.jks -storepass
supersecret
Certificate stored in file <jon.cer>
[janstey@bender]$ keytool -importcert -noprompt -alias jon
 \
        -file jon.cer -keystore
cia_truststore.jks \
        -storepass supersecret
Certificate was added to keystore

```

Now before you can use the keys stored in the keystores in Camel, you need to load them into the registry. Fortunately, Camel has the `org.apache.camel.util.jsse.KeyStoreParameters` helper class so you don't need to do this by hand. In Java, using `KeyStoreParameters` looks like this:

```

@Override
protected JndiRegistry createRegistry() throws Exception {
    JndiRegistry registry = super.createRegistry();

    KeyStoreParameters keystore = new KeyStoreParameters();
    keystore.setPassword("supersecret");
    keystore.setResource("./cia_keystore.jks");
    registry.bind("keystore", keystore);

    KeyStoreParameters truststore = new
KeyStoreParameters();
    truststore.setPassword("supersecret");
    truststore.setResource("./cia_truststore.jks");
    registry.bind("truststore", truststore);
    return registry;
}

```

In Spring or Blueprint XML, you can do this just as easily:

```

<keyStoreParameters
xmlns="http://camel.apache.org/schema/spring"
    id="keystore" resource="./cia_keystore.jks"
password="supersecret"/>
<keyStoreParameters
xmlns="http://camel.apache.org/schema/spring"
    id="truststore" resource="./cia_truststore.jks"
password="supersecret"/>

```

You have the key and trust stores ready, so you can start signing and verifying messages.

SIGNING AND VERIFYING MESSAGES USING CAMEL-CRYPTO

The camel-crypto component provides two endpoints for signing and verifying messages. The URI format is as follows:

```
crypto:sign:endpointName[?options]  
crypto:verify:endpointName[?options]
```

Intuitively, you'd use `crypto:sign` to add a digital signature to the message. Using our previously generated keystore, this would look something like this:

```
from("direct:sign")  
    .to("crypto:sign://keystore?  
keyStoreParameters=#keystore  
&alias=jon&password=secret")  
...
```

In XML, this would look like the following:

```
<route>  
    <from uri="direct:sign"/>  
    <to uri="crypto:sign://keystore?  
keyStoreParameters=#keystore&#  
alias=jon&password=secret"/>  
    ...  
</route>
```

This would add a `CamelDigitalSignature` header (defined in the `org.apache.camel.component.crypto.DigitalSignatureConstants`.SIGNATURE constant) to the message with a value calculated using the default SHA1WithDSA algorithm and the private key with alias `jon` in the keystore loaded via `KeyStoreParameters` in the `#keystore` reference. Verifying this message later is again a simple step in your route:

```
from("direct:verify")  
    .toF("crypto:verify://keystore?  
keystore=%s&alias=%s&password=%s",  
        "#truststore", "jon", "secret")  
...
```

In XML DSL, this would look like the following:

```

<route>
    <from uri="direct:verify"/>
    <to uri="crypto:verify://keystore?
keyStoreParameters=#truststore&amp;
                                alias=jon&amp;
                                password=secret"/>
    ...
</route>

```

The only difference from before is that now you're using the verify endpoint and you're using the truststore instead of the keystore. On success, nothing will happen, and your route will continue as normal. But when something bad happens to your message in transit, the verify call will generate a `java.security.SignatureException`, which you can then handle via Camel's exception-handling facility. For example, let's try to simulate a man-in-the-middle type attack by modifying the message body before it gets to the verify step.

NOTE See https://en.wikipedia.org/wiki/Man-in-the-middle_attack for more information on this type of attack.

You'll use three routes to simulate this type of attack:

```

from("direct:sign")
    .toF("crypto:sign://keystore?
keystore=%s&alias=%s&password=%s",
        "#keystore", "jon", "secret")
    .to("mock:signed")
    .to("direct:mitm");

from("direct:mitm")
    .setBody().simple("I'm hacked!")
    .to("direct:verify");

from("direct:verify")
    .toF("crypto:verify://keystore?
keystore=%s&alias=%s&password=%s",
        "#truststore", "jon", "secret")
    .to("mock:verified");

```

Now when sending a message to the `direct:sign` endpoint

```
try {
    template.sendBody("direct:sign", "Hello World");
} catch (CamelExecutionException e) {
    assertInstanceOf(SignatureException.class,
e.getCause());
}
```

you can expect its body to be modified to “I’m hacked!” and then the verify step will fail. Notice that Camel throws CamelExecutionException with the cause being SignatureException.

You can run this example for yourself using the following Maven goals from the chapter14/payload directory:

```
mvn test -Dtest=MessageSigningWithKeyStoreParamsTest
mvn test -Dtest=SpringMessageSigningWithKeyStoreParamsTest
mvn test -Dtest=ManInTheMiddleTest
```

Now that you’ve stopped a man-in-the-middle attack using digital signatures, let’s see how to encrypt entire payloads.

14.3.2 PAYLOAD ENCRYPTION

Sometimes even viewing the contents of a message can be considered a security breach. For these cases, encrypting the entire payload can be useful. Encryption can be performed via symmetric or asymmetric cryptography. In *symmetric* cryptography, both the sender and receiver share a secret key. This is in contrast to digital signatures we looked at previously, where the sender and receiver had different but complementary keys (they were *asymmetric*). Because you just looked at an asymmetric example, let’s take a look at how a symmetric case would work.

NOTE Asymmetric encryption is handled with PGPDataformat. You can find more information on this data format on Camel’s website, at <http://camel.apache.org/crypto.html>.

Similar to when signing messages in Camel, the camel-crypto component is used. You first have to create the key that will feed into the cryptography algorithms. Symmetric cryptography needs a completely different type of key than what was used for the digital signatures, so you can't reuse those keys.

GENERATING AND LOADING SECRET KEYS

To generate a secret key using keytool, you can use the `-genseckekey` command:

```
keytool -genseckekey
    -alias ciasecrets
    -keypass secret
    -keystore cia_secrets.jceks
    -storepass supersecret
    -storetype JCEKS
```

Here you're creating a secret key with the alias `ciasecrets` and key password of `secret`. Note that you can't use the default store type anymore; you need to use `JCEKS` for storing secret keys. Finally, you specify `cia_secrets.jceks` as the keystore file with a password of `supersecret`.

As for loading the keystore in Camel, you need to load the keystores in the same way as you did for message signing, except you can't use the default keystore type anymore. You need to specify `JCEKS` for the type:

```
@Override
protected JndiRegistry createRegistry() throws Exception {
    JndiRegistry registry = super.createRegistry();

    KeyStoreParameters keystore = new KeyStoreParameters();
    keystore.setPassword("supersecret");
    keystore.setResource("./cia_secrets.jceks");
    keystore.setType("JCEKS");

    KeyStore store = keystore.createKeyStore();
    secretKey = store.getKey("ciasecrets",
    "secret".toCharArray());
    registry.bind("secretKey", secretKey);
    return registry;
```

```
}
```

Here you use `KeyStoreParameters` again to do the grunt work of loading the keystore, but you're adding the secret key to the registry instead of `KeyStoreParameters`.

ENCRYPTING AND DECRYPTING PAYLOADS USING CAMEL-CRYPTO

Payload encryption in camel-crypto is implemented using data formats. We looked at data formats in chapter 3. Basically, using the `marshal` DSL method will encrypt the payload, and `unmarshal` will decrypt. Before you can call either `marshal` or `unmarshal`, you need to create the data format:

```
CryptoDataFormat crypto = new CryptoDataFormat("DES",  
secretKey);
```

Here you specify an encryption algorithm of Digital Encryption Standard (DES) and the secret key you created and loaded previously.

TIP All possible cipher algorithms are listed in the Java Cryptography Architecture Standard Algorithm Name Documentation. See <https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#Cipher>.

Now you can use the data format in a route like this:

```
from("direct:start")  
    .marshal(crypto)  
    .to("mock:encrypted")  
    .unmarshal(crypto)  
    .to("mock:unencrypted");
```

In XML DSL, this would look like the following:

```
<bean id="secretKey"  
      class="camelaction.SpringMessageEncryptionTest"  
      factory-method="loadKey"/>
```

```

<camelContext
    xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
        <crypto id="myCrypto" algorithm="DES"
keyRef="secretKey"/>
    </dataFormats>
    <route>
        <from uri="direct:start"/>
        <marshal ref="myCrypto"/>
        <to uri="mock:encrypted"/>
        <unmarshal ref="myCrypto"/>
        <to uri="mock:unencrypted"/>
    </route>
</camelContext>

```

At the `mock:encrypted` endpoint, the payload is encrypted, and at the `mock:unencrypted` endpoint, the payload is back to plain text. This route demonstrates how easily you can encrypt and decrypt a payload in Camel. After you've loaded your key and set up your data format, that's all there is to it.

You can run this example for yourself using the following Maven goals from the chapter14/payload directory:

```

mvn test -Dtest=MessageEncryptionTest
mvn test -Dtest=SpringMessageEncryptionTest

```

Another option to ensure that your data is encrypted outside your Camel route is to use transport security, which is all about setting up the TLS protocol on the underlying transport.

14.4 Transport security

Many components in Camel allow you to configure TLS settings for the transport type being offered. In Java, TLS is typically configured using the Java Secure Socket Extension (JSSE) API provided with the JRE. But not all third-party libraries that Camel integrates with use JSSE in a common way. To make this a bit easier, Camel provides a layer on top of this, called the JSSE Utility. At the time of writing, many Camel components use this utility:

- camel-ahc
- camel-apns
- camel-box
- camel-cometd
- camel-ftp
- camel-http and camel-http4
- camel-irc
- camel-jetty9
- camel-linkedin
- camel-mail
- camel-mina2
- camel-netty and camel-netty4
- camel-olingo2 and camel-olingo4
- camel-restlet
- camel-salesforce
- camel-spring-ws
- camel-undertow
- camel-websocket

The central class you'll be using from this utility is `org.apache.camel.util.jsse.SSLContextParameters`. That's what you'll be passing into the Camel components to configure the TLS settings. You can configure this via pure Java or in Spring or Blueprint XML. For example, in Spring, a possible `SSLContextParameters` configuration looks like this:

```
<sslContextParameters id="sslContextParameters">  
    xmlns="http://camel.apache.org/schema/spring">  
        <keyManagers keyPassword="secret">
```

```

        <keyStore resource="./cia_keystore.jks"
password="supersecret"/>
    </keyManagers>
    <trustManagers>
        <keyStore resource="./cia_truststore.jks"
password="supersecret"/>
    </trustManagers>
</sslContextParameters>
```

Here you have a top-level `sslContextParameters` element and then `keyManagers` and `trustManagers` children. These configure the keystore and truststore files, respectively, that you created in section 14.3.1. From here, you can then reference the `sslContextParameters` bean in your endpoint URI:

```

<route>
    <from uri="jetty:https://localhost:8080/early?
sslContextParameters=#sslContextParameters"/>
    <transform>
        <constant>Hi</constant>
    </transform>
</route>
```

With this route, you now have an HTTPS endpoint up and running. In Java, things get more verbose, but it's mostly the same:

```

@Override
protected JndiRegistry createRegistry() throws Exception {
    KeyStoreParameters ksp = new KeyStoreParameters();
    ksp.setResource("./cia_keystore.jks");
    ksp.setPassword("supersecret");
    KeyManagersParameters kmp = new
KeyManagersParameters();
    kmp.setKeyPassword("secret");
    kmp.setKeyStore(ksp);

    KeyStoreParameters tsp = new KeyStoreParameters();
    tsp.setResource("./cia_truststore.jks");
    tsp.setPassword("supersecret");
    TrustManagersParameters tmp = new
TrustManagersParameters();
    tmp.setKeyStore(tsp);

    SSLContextParameters sslContextParameters = new
SSLContextParameters();
```

```

    sslContextParameters.setKeyManagers(kmp);
    sslContextParameters.setTrustManagers(tmp);

    JndiRegistry registry = super.createRegistry();
    registry.bind("ssl", sslContextParameters);

    return registry;
}

```

The Java route is also similar:

```

from("jetty:https://localhost:8080/early?
sslContextParameters=#ssl")
    .transform().constant("Hi");

```

When calling these HTTPS endpoints within Camel, you need to also provide `sslContextParameters`, which contains a trusted certificate. For the purposes of the example, you can reuse the server `sslContextParameters`. The URI syntax is exactly the same for the producer in this case:

```

@Test
public void testHttps() throws Exception {
    String reply = template.requestBody(
        "jetty:https://localhost:8080/early?
sslContextParameters=#ssl",
        "Hi Camel!", String.class);
    assertEquals("Hi", reply);
}

```

If you didn't provide any `sslContextParameters` with a valid trust store, you should expect Camel to throw an execution exception, because the server wouldn't let you connect:

```

@Test(expected = CamelExecutionException.class)
public void testHttpsNoTruststore() throws Exception {
    String reply = template.requestBody(
        "jetty:https://localhost:8080/early", "Hi Camel!",
String.class);
    assertEquals("Hi", reply);
}

```

You can try this for yourself using the following Maven goals from the chapter14/transport directory:

```
mvn test -Dtest=HttpsTest  
mvn test -Dtest=SpringHttpsTest
```

14.4.1 DEFINING GLOBAL SSL CONFIGURATION

If you use the same SSL configuration in most of your routes, it makes sense to define this configuration globally. You can do this by first setting the global `SSLContextParameters` on `CamelContext`:

```
CamelContext context = super.createCamelContext();  
context.setSSLContextParameters(createSSLContextParameters());
```

The `createSSLContextParameters` method in this case creates an `SSLContextParameters` object:

```
private SSLContextParameters createSSLContextParameters() {  
    KeyStoreParameters ksp = new KeyStoreParameters();  
    ksp.setResource("./cia_keystore.jks");  
    ksp.setPassword("supersecret");  
    KeyManagersParameters kmp = new  
    KeyManagersParameters();  
    kmp.setKeyPassword("secret");  
    kmp.setKeyStore(ksp);  
  
    KeyStoreParameters tsp = new KeyStoreParameters();  
    tsp.setResource("./cia_truststore.jks");  
    tsp.setPassword("supersecret");  
    TrustManagersParameters tmp = new  
    TrustManagersParameters();  
    tmp.setKeyStore(tsp);  
  
    SSLContextParameters sslContextParameters = new  
    SSLContextParameters();  
    sslContextParameters.setKeyManagers(kmp);  
    sslContextParameters.setTrustManagers(tmp);  
  
    return sslContextParameters;  
}
```

You can then enable this global SSL config for each component you're using. In this case, the SSL examples in this chapter use the Jetty component so you can enable global SSL like so:

```
((SSLContextParametersAware) context.getComponent("jetty"))  
.setUseGlobalSslContextParameters(true);
```

After this, you no longer have to reference the `SSLContextParameters` in our endpoint URIs. The example route you used previously simplifies down to this:

```
from("jetty:https://localhost:8080/early")
    .transform().constant("Hi");
```

You can try this for yourself using the following Maven goal from the `chapter14/transport` directory:

```
mvn test -Dtest=GlobalSSLContextParametersTest
```

Now that we've covered security at the transport level, let's move back into Camel and look at implementing authentication and authorization in your routes.

14.5 Route authentication and authorization

The process of authentication and authorization are separate concepts, but they always go together. *Authentication* is the first step, whereby a user proves who they claim to be to a system using, typically, a username and password. Next, after successful authentication, *authorization* is the process whereby the system checks what you're allowed to do. Camel supports both of these processes within a route so you can control who is allowed to use a particular route. There are two implementations of route security: one using Apache Shiro, and another using Spring Security. Both frameworks provide a means of authenticating and authorizing a user. They do more than that, but from a Camel point of view, we're using only these features. You can find out more about the respective projects on their websites:

- *Spring Security*—<http://projects.spring.io/spring-security>
- *Apache Shiro*—<http://shiro.apache.org>

Spring Security seems to be the most popular and active project of the two, so we focus on that in the examples of this section. For more information on Camel's Shiro support, check out the

website docs at <http://camel.apache.org/shiro-security.html>.

To get started using Spring Security in Camel, you need to add a dependency on the camel-spring-security module to your Maven POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-security</artifactId>
  <version>2.20.1</version>
</dependency>
```

This brings in the necessary Spring Security JARs transitively, so you don't need to add them to your POM as well. If you're curious, though, or are looking at the reference documentation, Camel uses the spring-security-core and spring-security-config modules. Several more modules are available to support things like LDAP and ACLs. These additional modules are beyond the scope of this section, so to find out more, check out the Spring Security project's website.

Route security in Camel is handled via a *policy*, a class that implements org.apache.camel.spi.Policy. A route protected by a policy looks like this:

```
<route>
  <from uri="direct:start"/>
  <to uri="mock:unsecure"/>
  <policy ref="admin">
    <to uri="mock:secure"/>
  </policy>
</route>
```

Here you're referencing a policy defined by the `admin` bean. Note that the whole route isn't protected by the policy—only what's inside the policy element. You must take care to place the secured calls inside the policy element. A *policy* is a generic Camel concept, so you haven't really done any securing yet. Let's take a look at that `admin` bean to see how it's helping to secure your route:

```
<authorizationPolicy
  xmlns="http://camel.apache.org/schema/spring-security"
```

```
    id="admin" access="ROLE_ADMIN"
    authenticationManager="authenticationManager"
    accessDecisionManager="accessDecisionManager"/>
```

This `authorizationPolicy` element is syntactic sugar for the `org.apache.camel.component.spring.security.SpringSecurityAuthorizationPolicy` class, which implements the `Policy` interface. You could have created a `SpringSecurityAuthorizationPolicy` bean here just the same.

Let's go over what the attributes of `authorizationPolicy` are doing. First, the `accessDecisionManager` attribute specifies the `org.springframework.security.access.AccessDecisionManager` that you'll be using. This is the class that will be deciding whether the user will be authorized to use the route. The `access` attribute specifies the authority name passed to the `AccessDecisionManager`. This is most often a role name (a string beginning with the prefix `ROLE_`). Finally, the `authenticationManager` attribute specifies the `org.springframework.security.authentication.AuthenticationManager` bean that you'll be using to authenticate the user.

14.5.1 CONFIGURING SPRING SECURITY

Outside Camel, you need to configure two required Spring Security beans: `AccessDecisionManager` and `AuthenticationManager`. To configure these, you need to add an extra XML namespace to your Spring XML:

```
xmlns:spring-
security="http://www.springframework.org/schema/security"
xsi:schemaLocation="
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-
security.xsd
..."
```

From here, you can then define `AuthenticationManager`:

```
<spring-security:authentication-manager
alias="authenticationManager">
    <spring-security:authentication-provider
        user-service-ref="userDetailsService"/>
</spring-security:authentication-manager>
```

```
<spring-security:user-service id="userDetailsService">
    <spring-security:user name="jon"
        password="secret" authorities="ROLE_USER"/>
    <spring-security:user name="claus"
        password="secret" authorities="ROLE_USER,
ROLE_ADMIN"/>
</spring-security:user-service>
```

This is probably the simplest `AuthenticationManager` configuration you can get. All you do here is check that a password matches the one provided for a particular user. The usernames, passwords, and roles are all defined within the XML snippet. In a real-world deployment, you'd probably have users checked via JAAS or by querying an LDAP database. It's far easier to talk about an XML snippet over an LDAP server configuration! In the preceding snippet, you're defining two users: `jon` and `claus`. Each has a different set of roles. Only `claus` has the `admin` role. Recall that before, your Camel `authorizationPolicy` had an `access` parameter defined as `ROLE_ADMIN`, which means only users with the `admin` role will be allowed to access the route inside the `policy` element. But how is this role checking defined? The answer is `AccessDecisionManager`. For our role-checking example, you can use the following `AccessDecisionManager` configuration:

```
<bean id="accessDecisionManager"
    class="org.springframework.security.access.vote.Affirmative
    Based">
    <property name="decisionVoters">
        <list>
            <bean
                class="org.springframework.security.access.vote.RoleVoter"/>
        </list>
    </property>
</bean>
```

The concrete `AccessDecisionMaker` you're using here, `AffirmativeBased`, will grant access if one of the decision voters succeeds. You have only one decision voter configured here, so

it's one vote or nothing. The sole vote is coming from `RoleVoter`, which will grant access only if the user has the specified role (in this case, it's `ROLE_ADMIN`).

Now your route is secured! But you need to do a little more work to use it now. Trying to access it at this point will certainly fail, unless you pass in the proper credentials.

`SpringSecurityAuthorizationPolicy` looks for `Exchange.AUTHENTICATION` headers (`camelAuthentication`) for the `javax.security.auth.Subject` it will be authenticating. You can easily create this header from a username and password:

```
private void sendMessageWithAuth(String uri, String body,
                                String username, String
password) {
    Authentication authToken =
        new UsernamePasswordAuthenticationToken(username,
password);

    Subject subject = new Subject();
    subject.getPrincipals().add(authToken);

    template.sendBodyAndHeader(uri, body,
        Exchange.AUTHENTICATION, subject);
}
```

Let's look at a test in action:

```
@Test
public void testAdminOnly() throws Exception {

    getMockEndpoint("mock:secure").expectedBodiesReceived("Davs
Claus!");

    getMockEndpoint("mock:unsecure").expectedBodiesReceived("Da
vs Claus!"
        , "Hello Jon!");
    sendMessageWithAuth("direct:start", "Davs Claus!",
"claus", "secret");
    try {
        sendMessageWithAuth("direct:start",
                "Hello Jon!", "jon", "secret");
    } catch (CamelExecutionException e) {

        assertEquals(CamelAuthorizationException.class,
```

```
        e.getCause());
    }

    assertMockEndpointsSatisfied();
}
```

Here you're sending two messages: one using claus's credentials, and another using jon's. Because your policy requires the `admin` role, only claus's message gets through. On sending jon's credentials, Camel throws `CamelExecutionException`, with the cause being `CamelAuthorizationException`.

You can run this example for yourself using the following Maven goals from the chapter14/route directory:

```
mvn test -Dtest=SpringSecurityTest
```

This covers the basics of using Spring Security in Camel for route authentication and authorization. For more information, see the Camel website at <http://camel.apache.org/spring-security.html>.

14.6 Summary and best practices

This concludes our whirlwind tour of Camel's security features. As with many other areas of Camel, you can implement security in many ways. Camel doesn't force you to use a set library for security.

Let's recap some key ideas from this chapter:

- *Secure only what you need to*—Of the four types of security configuration in Camel, you often don't need to use all of them, or even more than one. For instance, you probably don't need to encrypt a payload when you have transport security enabled, as this will also encrypt the payload for you. Also, extra unnecessary security configuration takes away from the readability of your Camel applications.
- *Camel is unsecured by default*—Probably the most important point is that Camel has no security settings turned on by default. This is great for development, but before your application is

deployed in the real world, you'll most likely need some form of security enabled.

Speaking of deploying Camel applications in the real world, in chapter 15 we'll be talking all about running and deploying Camel in various environments.

Part 5

Running and managing Camel

With all the preceding Camel concepts under your belt, you may feel as if you can tackle any integration problem with Camel. There's still more to cover, though. Deployment is a topic you'll need to read about before your Camel application is in production. Camel was designed as a framework, and as such has virtually unlimited deployment possibilities. In chapter 15, we'll discuss some of the most popular deployment options for Camel. We'll also cover the various ways of starting and stopping Camel in a safe manner.

Once your application is running, you'll need to know how to manage it and monitor its operations. Chapter 16 discusses topics in this category, such as viewing the Camel logs, controlling Camel with Jolokia and JMX, and extending the notification mechanism in Camel so it works with your own custom monitoring application.

15

Running and deploying Camel

This chapter covers

- Starting and stopping Camel safely
- Adding and removing routes at runtime
- Deploying Camel
- Running standalone
- Running in web containers
- Running in Java EE servers
- Running with OSGi
- Running with CDI

In chapter 14, you learned all about securing Camel. We'll now shift focus to another topic that's important to master: running and deploying Camel applications.

We'll begin with starting Camel. You need to fully understand how to start, run, and shut down Camel reliably and safely, which is imperative in a production environment. We'll also review various options you can use to tweak the way Camel and routes are started. We'll continue on this path, looking at how to dynamically start and stop routes at runtime. Your applications won't run forever, so we'll spend time focusing on how to shut down Camel safely.

The other part of this chapter covers various strategies for deploying Camel. We'll take a look at five common runtime environments supported by Camel. Two other popular runtimes, Spring Boot and WildFly Swarm, are covered in chapter 7, where we discuss microservices.

As we discuss these topics, we'll work through an example involving Rider Auto Parts: you've been asked to help move a recently developed application safely into production. The application receives inventory updates from suppliers, provided via a web service or files. [Figure 15.1](#) shows a high-level diagram of the application.

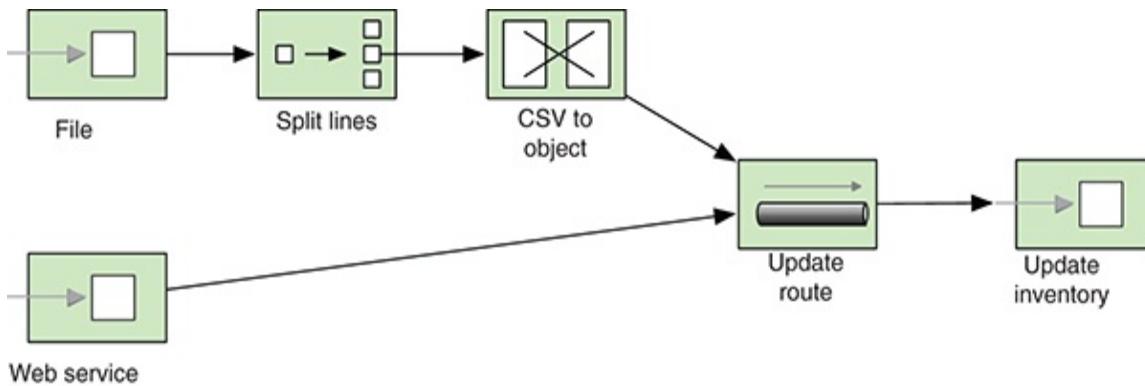


Figure 15.1 A Rider Auto Parts application accepting incoming inventory updates from either files or a web service

15.1 Starting Camel

In chapter 1, you learned how to download, install, and run Camel. That works well in development, but the game plan changes when you take an application into production.

Starting up a Camel application in production is harder than you might think because the route order may have to be arranged in a certain way to ensure a reliable startup. It's critical that the operations staff can safely manage the application in their production environment.

Let's look at how Camel starts.

15.1.1 HOW CAMEL STARTS

Camel doesn't start magically by itself. Often it's the server (container) that Camel is running inside that invokes the `start` method on `CamelContext`, starting up Camel. This is also what you saw in chapter 1, where you used Camel inside a standalone Java application. A standalone Java application isn't the only deployment choice; you can also run Camel inside a container such as Spring, CDI, or OSGi.

Regardless of which container you use, the same principle applies: the container must prepare and create an instance of `CamelContext` up front, before Camel can be started, as illustrated in [figure 15.2](#).

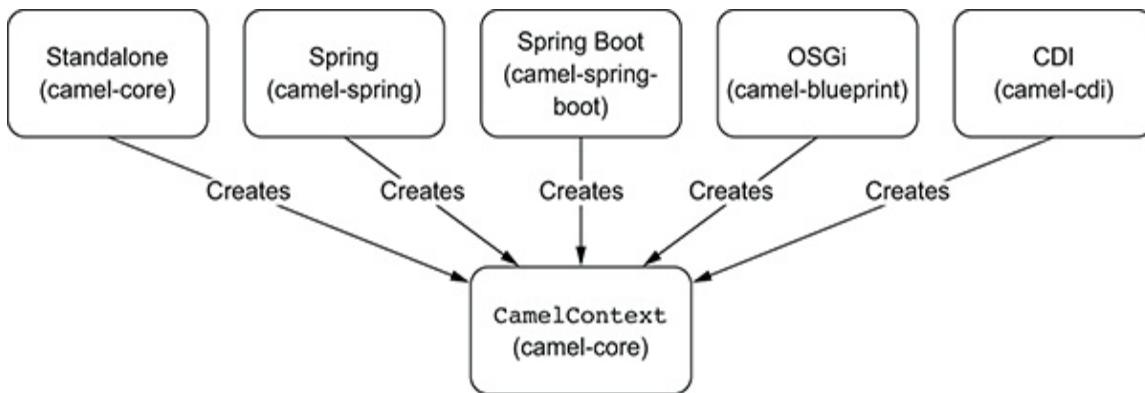


Figure 15.2 Using Camel with containers often requires the container in question to prepare and create `CamelContext` up front before it can be started.

Because Spring is a common container, we'll outline how Spring and Camel work together to prepare `CamelContext`.

PREPARING CAMELCONTEXT IN A SPRING CONTAINER

Spring allows third-party frameworks to integrate seamlessly with Spring. To do this, the third-party frameworks must provide `org.springframework.beans.factory.xml.NamespaceHandler`, which is the extension point for using custom namespaces in Spring XML files. Camel provides `CamelNamespaceHandler`.

When using Camel in the Spring XML file, you define the `<camelContext>` tag as follows:

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

The `http://camel.apache.org/schema/spring` namespace is the Camel custom namespace. To let Spring know about this custom namespace, Camel provides a `META-INF/spring.handlers` file, where the namespace is mapped to the class implementation:

```
http://camel.apache.org/schema/spring=org.apache.camel.spring.handler.CamelNamespaceHandler
```

`CamelNamespaceHandler` is then responsible for parsing the XML and delegating to other factories for further processing. One of these factories is `camelContextFactoryBean`, which is responsible for creating the `camelContext` that essentially is your Camel application.

Preparing CamelContext in Spring Boot

When Camel is used with Spring Boot, then Camel is prepared using Spring Boot *auto configuration*, which is the standard way on Spring Boot. This is implemented in the `org.apache.camel.spring.boot.CamelAutoConfiguration` class in the `camel-spring-boot` JAR.

When Spring is finished initializing, it signals to third-party frameworks that they can start by broadcasting the `ContextRefreshedEvent` event.

STARTING CAMEL CONTEXT

At this point, `CamelContext` is ready to be started. What happens next is the same regardless of which container or deployment option you're using with Camel. [Figure 15.3](#) shows a flow diagram of the startup process.

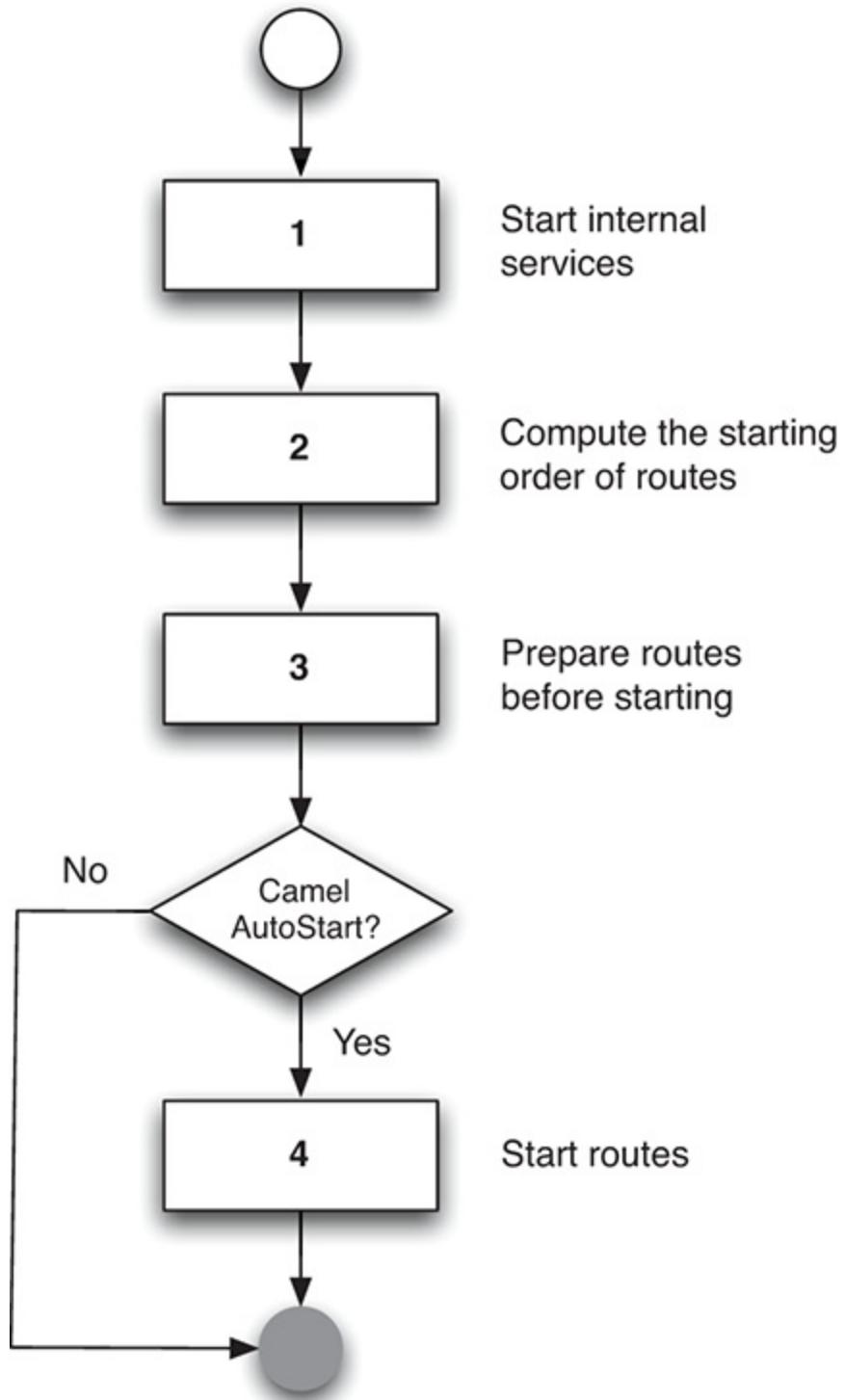


Figure 15.3 Flow diagram showing how Camel starts by starting internal services, computing the starting order of routes, and preparing and starting the routes

CamelContext is started by invoking its start method. [Figure 15.3](#)

shows the following four steps:

1. *Start internal services*—Prepares and starts internal services used by Camel, such as the type-converter mechanism.
2. *Compute starting order*—Computes the order in which the routes should be started. By default, Camel starts up all the routes in the order they're defined in the XML files or the Java RouteBuilder classes. Section 15.1.3 covers how to configure the order of routes.
3. *Prepare routes*—Prepares the routes before they're started.
4. *Start routes*—If AutoStartup is true, starts the routes by starting the consumers, which opens the gates to Camel and lets the messages start to flow in.

After step 4, Camel writes a message to the log indicating that it's been started and that the startup process is complete. In some cases, you may need to influence how Camel is started, so let's look at that now.

15.1.2 CAMEL STARTUP OPTIONS

Camel offers various startup options. For example, you may have a maintenance route that shouldn't be autostarted on startup. You may also want to enable tracing on startup to let Camel log traces of messages being routed.

Table 15.1 lists all the options that influence startup. These options can be divided into two kinds. The first four are related to startup and shutdown, and the remainder are miscellaneous options. We'll explain the miscellaneous options first and then turn our attention to the startup and shutdown options.

Table 15.1 Camel startup options

Option	Description
AutoStartup	Indicates whether the route should be started automatically when Camel starts. This option is enabled by default.

StartUpOrder	Dictates the order in which the routes should be started when Camel starts. We cover this in section 15.1.3.
ShutdownRoute	Configures whether the route in question should stop immediately or defer while Camel is shutting down. We cover shutdown in section 15.3.
ShutdownRunningTask	Controls whether Camel should continue to complete pending running tasks at shutdown or stop immediately after the current task is complete. We cover shutdown in section 15.3.
Tracing	Traces how an exchange is being routed within that particular route. This option is disabled by default. Tracing is covered in chapter 16.
Delay	Sets a delay in milliseconds that slows the processing of a message. You can use this during debugging to reduce how quickly Camel routes messages, which may help you track what happens when you watch the logs. This option is disabled by default.
MessageHistory	Stores the history of an exchange as it's being routed within a particular route. This option is enabled by default. Chapter 11, section 11.3.3 has sample output of the MessageHistory option in action.
HandleFault	Turns fault messages into exceptions. This isn't a typical thing to do in a pure Camel application, but when using SOAP faults, you'll need to set this option to let the Camel error handler react to faults. This option is disabled by default. We cover this in more detail shortly.
StreamCaching	Caches streams that otherwise couldn't be accessed multiple times. You may want to use this when you use redelivery during error handling, which requires being able to read the stream multiple times. This option is disabled by default. We cover this in more detail shortly.
AllowUseOriginalMessage	Tells Camel to make the original message available in error handlers. This option is enabled by default. If you don't need access to the original message in your error handlers, consider turning this option off; it may improve performance.
LogExhaustedMessageBody	Sets whether to log the full message body in the message history.

LogMask	Determines whether logging (log component, Log EIP, tracer, and so on) should mask sensitive information such as passwords. This option is disabled by default.
---------	---

CONFIGURING STREAMCACHING

The miscellaneous options, such as the Tracing option, are often used during development to turn on additional logging, which we cover in detail in the next chapter. Or you may need to turn on stream caching if you use Camel with other stream-centric systems. For example, to enable stream caching, you can do the following with the Java DSL:

```
public class MyRoute extends RouteBuilder {
    public void configure() throws Exception {
        context.setStreamCaching(true);
        from("cxft:bean:inventoryEndpoint?
dataFormat=PAYLOAD")
        ...
    }
}
```

The same example using XML DSL looks like this:

```
<camelContext streamCache="true"
    xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="cxft:bean:inventoryEndpoint?
dataFormat=PAYLOAD"/>
        ...
    </route>
</camelContext>
```

All the options from table 15.1 can be scoped at either the context or route level. The preceding stream cache example was scoped at the context level. You could also configure it on a particular route:

```
public class MyRoute extends RouteBuilder {
    public void configure() throws Exception {
        from("cxft:bean:inventoryEndpoint?
dataFormat=PAYLOAD")
        .streamCaching()
```

```
    ...
}
```

You can configure route-scoped stream caching in XML as follows:

```
<camelContext
  xmlns="http://camel.apache.org/schema/spring">
  <route streamCache="true">
    <from uri="cxf:bean:inventoryEndpoint?
dataFormat=PAYLOAD"/>
    ...
  </route>
</camelContext>
```

NOTE Java DSL uses the syntax noXXX to disable an option, such as noStreamCaching or noTracing.

You need to know one last detail about the context and route scopes. The context scope is used as a fallback if a route doesn't have a specific configuration. The idea is that you can configure the default setting on the context scope and then override when needed at the route scope. For example, you could enable tracing on the context scope and then disable it on the routes you don't want traced.

CONFIGURING HANDLEFAULT

In the preceding example, you routed messages using the CXF component. The `HandleFault` option is used to control whether Camel error handling should react to SOAP faults.

Suppose sending to the `cxf:bean:inventoryEndpoint? dataFormat=PAYLOAD` endpoint fails with a fault. Without `HandleFault` enabled, the fault would be propagated back to the consumer. By enabling `HandleFault`, you can let the Camel error handler react when faults occur.

The following code shows how to let the `DeadLetterChannel`

error handler save failed messages to files in the error directory:

```
public class MyRoute extends RouteBuilder {  
    public void configure() throws Exception {  
        errorHandler(deadLetterChannel("file:errors"));  
        from("direct:start")  
            .streamCaching().handleFault()  
            .to("cxft:bean:inventoryEndpoint?  
dataFormat=PAYLOAD");  
    }  
}
```

The equivalent example in XML is as follows:

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
    <errorHandler id="EH" type="DeadLetterChannel"  
                  deadLetterUri="file:errors"/>  
    <route streamCache="true" errorHandlerRef="EH"  
handleFault="true">  
        <from uri="direct:start"/>  
        <to uri="cxft:bean:inventoryEndpoint?  
dataFormat=PAYLOAD"/>  
    </route>  
</camelContext>
```

We'll now look at how to control the ordering of routes.

15.1.3 ORDERING ROUTES

The order in which routes are started and stopped can become important when they're highly interdependent. For example, you may have reusable routes that must be started before being used by other routes. Also, routes that immediately consume messages that are bound for other routes may have to be started later to ensure that the other routes are ready in time.

To control the startup order of routes, Camel provides two options: `AutoStartup` and `StartupOrder`. The former dictates whether the routes should be started. The latter is a number that dictates the order in which the routes should be started.

USING STARTUPORDER TO CONTROL ORDERING OF ROUTES

Let's return to our Rider Auto Parts example, outlined at the beginning of the chapter. [Figure 15.4](#) shows the high-level diagram again, this time numbering the three routes in use: 1, 2, and 3.

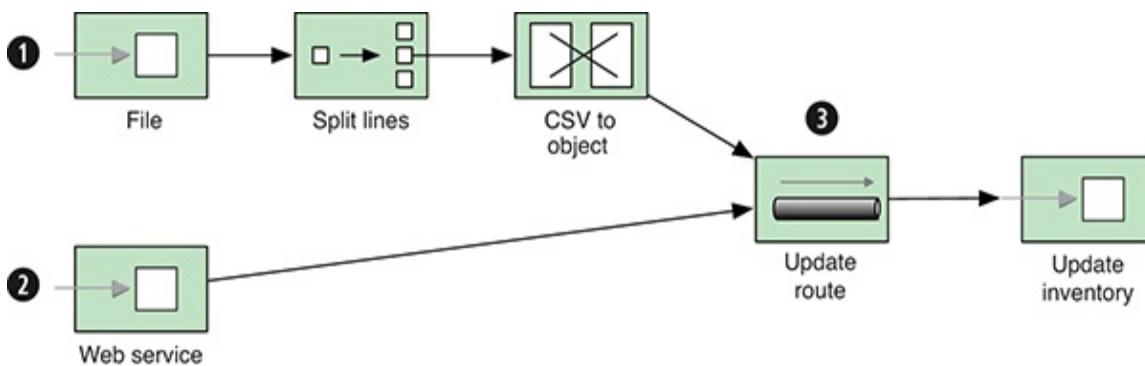


Figure 15.4 Camel application with two input routes, 1 and 2, that depend on a common route, 3

The file-based route ① polls incoming files and splits each line in the file. The lines are then converted to an internal `camelaction.inventory.UpdateInventoryInput` object, which is sent to route 3.

The web service route ② is much simpler, because incoming messages are automatically converted to the `UpdateInventoryInput` object. The web service endpoint is configured to do this.

The update route ③ is a common route that's reused by the first two routes.

You now have a dependency among the three routes. Routes 1 and 2 depend on route 3, and that's why you can use `StartupOrder` to ensure that the routes are started in correct order.

The following listing shows the Camel routes with the `StartupOrder` options in boldface.

Listing 15.1 Starting routes in a specific order

```
public class InventoryRoute extends RouteBuilder {  
    @Override
```

```

public void configure() throws Exception {
    from("cxf:bean:inventoryEndpoint")
        .routeId("webservice").startupOrder(3)
        .to("direct:update")
        .transform().method("inventoryService",
"replyOk");
    from("file://target/inventory/updates")
        .routeId("file").startupOrder(2)
        .split(body().tokenize("\n"))
            .convertBodyTo(UpdateInventoryInput.class)
            .to("direct:update")
        .end();
    from("direct:update")
        .routeId("update").startupOrder(1)
        .to("bean:inventoryService?
method=updateInventory");
}
}

```

[Listing 15.1](#) shows how easy it is in the Java DSL to configure the order of the routes using the startupOrder method. [Listing 15.2](#) shows the same example using the XML DSL.

NOTE In [listing 15.1](#), routeId is used to assign each route a meaningful name, which will then show up in the management console or in the logs. If you don't assign an ID, Camel will auto-assign an ID using the pattern route1, route2, and so forth.

Listing 15.2 XML DSL version of listing 15.1

```

<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <route id="webservice" startupOrder="3">
        <from uri="cxf:bean:inventoryEndpoint"/>
        <to uri="direct:update"/>
        <transform>
            <method bean="inventoryService"
method="replyOk"/>
        </transform>
    </route>
    <route id="file" startupOrder="2">
        <from uri="file://target/inventory/updates"/>

```

```

<split>
    <tokenize token="\n"/>
    <convertBodyTo

type="camelaction.inventory.UpdateInventoryInput"/>
        <to uri="direct:update"/>
    </split>
</route>
<route id="update" startupOrder="1">
    <from uri="direct:update"/>
    <to uri="bean:inventoryService?
method=updateInventory"/>
</route>
</camelContext>

```

Notice that the numbers 1, 2, and 3 are used to dictate the order of the routes. Let's take a moment to see how this works in Camel.

How **STARTUPORDER** WORKS

The `startupOrder` option in Camel works much like the `load-on-startup` option for Java servlets. As with servlets, you can specify a positive number to indicate the order in which the routes should be started.

The numbers don't have to be consecutive. For example, you could use the numbers 5, 20, and 42 instead of 1, 2, and 3. All that matters is that the numbers must be unique.

You can also omit assigning `startupOrder` to some of the routes. In that case, Camel will automatically assign these routes a unique number starting with 1,000 upward. The numbers from 1 to 999 are intended for Camel users, and the numbers from 1,000 upward are reserved by Camel.

TIP The routes are stopped in the reverse order in which they were started.

In practice, you may not need to use `StartupOrder` often. Camel is

much more careful than it used to be when starting routes. In fact, in this example, if you let Camel determine the start order, it would likely be okay. The potential for issues comes most often if your endpoints are getting hammered with incoming messages while Camel is restarting. Then there's a chance a message could get in before all routes are operational. If that's the case, you may see an error like this:

```
org.apache.camel.component.direct.DirectConsumerNotAvailableException: No consumers available on endpoint:  
direct://update. Exchange[ID-ghost-42425-1490308252187-0-  
16]
```

The error indicates that the `update` route didn't start in time before someone sent a message to the `webservice` route.

NOTE The Camel team will improve on this for the upcoming Camel 2.21 release to let the direct component be smarter during startup and gracefully handle such a situation. Almost all the other components already handle interdependent startup ordering gracefully, so the need for explicit startup ordering will become less relevant in the future.

The book's source code contains this example in the `chapter15/startup` directory. You can try it using the following Maven goals:

```
mvn test -Dtest=InventoryJavaDSLTest  
mvn test -Dtest=InventorySpringXMLTest
```

You've now learned to control the order in which routes are started. Let's move on and take a look at how to omit starting certain routes and start them on demand later, at runtime.

15.1.4 DISABLING AUTOSTARTUP

Table 15.1 listed the `AutoStartup` option, which is used to specify whether a given route should be automatically started when Camel starts. Sometimes you may not want to start a route

automatically. You may want to start it on demand at runtime to support business cases involving human intervention.

At Rider Auto Parts, there has been a demand to implement a manual process for updating the inventory based on files. As usual, you've been asked to implement this in the existing application depicted in [figure 15.4](#).

You come up with the following solution: add a new route to the existing routes in [listing 15.1](#). The new route listens for files being dropped in the manual directory, and uses these files to update the inventory:

```
from("file://target/inventory/manual")
    .routeId("manual")
    .log("Doing manual update with file ${file:name}")
    .split(body().tokenize("\n"))
        .convertBodyTo(UpdateInventoryInput.class)
        .to("direct:update")
.end();
```

As you can see, the route is merely a copy of the file-based route in [listing 15.1](#). Unfortunately, your boss isn't satisfied with the solution. The route is always active, so if someone accidentally drops a file into the manual folder, it would be picked up.

To solve this problem, you use the `AutoStartup` option to disable the route from being activated on startup:

```
from("file://target/inventory/manual")
    .routeId("manual").noAutoStartup()
    .log("Doing manual update with file ${file:name}")
    .split(body().tokenize("\n"))
        .convertBodyTo(UpdateInventoryInput.class)
        .to("direct:update")
.end();
```

Notice that you use `noAutoStartup()` to disable route startup. You could have used `autoStartup(false)`, `autoStartup("false")`, or even `autoStartup("${{startupRouteProperty}}")`, where a property is referenced. In the XML DSL, there's no `noAutoStartup()`, but you can use `autoStartup="false"` or the other alternatives mentioned.

Now that the route isn't started by default, you can start the route when a manual file is meant to be picked up. This can be done using a management console, such as JConsole, to manually start the route, waiting until the file has been processed, and manually stopping the route again.

You've now learned how to configure Camel with various options that influence how it starts up. The next section presents various ways of programmatically controlling the lifecycle of routes at runtime.

15.2 Starting and stopping routes at runtime

In chapter 16, you'll learn how to use management tooling, designed for operations staff, to start and stop routes at runtime. Being able to programmatically control routes at runtime is also desirable. For example, you might want business logic to automatically turn routes on or off at runtime. This section explains how to do this.

You can start and stop routes at runtime in several ways, including these:

- *Using CamelContext*—By invoking the `startRoute`/`startAllRoutes` and `stopRoute` methods.
- *Using RoutePolicy*—By applying a policy to routes that Camel enforces automatically at runtime.
- *Using JMX*—By obtaining the `ManagedRoute` MBean for the particular routes and invoking its `start` or `stop` methods. If you've enabled remote management, you can control the routes from another machine.
- *Using the ControlBus component*—The control bus allows you to manage routes at runtime using regular messaging instead of a special SPI. This is also discussed in chapter 16, in section 16.4.3.

The use of JMX is covered in chapter 16; we discuss using

`CamelContext`, `ControlBus`, and `RoutePolicy` in this chapter.

15.2.1 USING CAMELCONTEXT TO START AND STOP ROUTES AT RUNTIME

`CamelContext` provides methods to easily start and stop routes. To illustrate this, we'll continue with the Rider Auto Parts example from section 15.1.4. About a month into production with the new route, one of the operations staff forgets to manually stop the route after use, as he was supposed to. Not stopping the route leads to a potential risk because files accidentally dropped into the manual directory will be picked up by the route.

You're again summoned to remedy this problem, and you quickly improve the route with the two changes shown in bold in the following listing.

Listing 15.3 After a file has been processed, the route is stopped

```
from("file://target/inventory/manual?maxMessagesPerPoll=1")
    .routeId("manual").noAutoStartup()
    .log("Doing manual update with file ${file:name}")
    .split(body()).tokenize("\n")
        .convertBodyTo(UpdateInventoryInput.class)
        .to("direct:update")
    .end()
    .process(new Processor() {
        public void process(Exchange exchange) throws
Exception {
            exchange.getContext().getInflightRepository()
                .remove(exchange, "manual");

            ExecutorService executor =
                getContext().getExecutorServiceManager()
                    .newSingleThreadExecutor(this,
"StopRouteManually");
            executor.submit(new Callable<Object>() {
                @Override
                public Object call() throws Exception {
                    log.info("Stopping route manually");
                    getContext().stopRoute("manual");
                    return null;
                }
            });
        }
    });
}
```

```
        }
    });
});
```

The first change uses the `maxMessagesPerPoll` option to tell the file consumer to pick up only one file at a time. The second change stops the route after that one file has been processed. This is done with the help of the inlined Processor, which can access `CamelContext` and tell it to stop the route by name. (`CamelContext` also provides a `startRoute` method for starting a route.) Before you stop the route, you must unregister the current exchange from the in-flight registry, which otherwise would prevent Camel from stopping the route, because it detects an exchange in progress. It's also important to call `stopRoute` from a thread separate from the current route. Older versions of Camel were less picky about this, but with the latest release, you need to use a separate thread.

The book's source code contains this example, which you can try from the `chapter15/startup` directory using the following Maven goal:

```
mvn test -Dtest=ManualRouteWithStopTest
```

Even though the fix to stop the route is simple, using the inlined processor at the end of the route isn't an optimal solution. It'd be better to keep the business logic separated from the stopping logic. That can be done with a feature called `OnCompletion`.

USING ONCOMPLETION

`OnCompletion` allows you to do *additional* routing after the original route is done. The classic example is to send an email alert if a route fails, but `onCompletion` has a broad range of uses.

Instead of using the inlined processor to stop the route, you can use `OnCompletion` in `RouteBuilder` to process the `StopRouteProcessor` class containing the logic to stop the route.

This is shown in bold in the following code:

```
public void configure() throws Exception {  
    onCompletion().process(new  
StopRouteProcessor("manual"));  
    from("file://target/inventory/manual?  
maxMessagesPerPoll=1")  
        .routeId("manual").noAutoStartup()  
        .log("Doing manual update with file ${file:name}")  
        .split(body().tokenize("\n"))  
            .convertBodyTo(UpdateInventoryInput.class)  
            .to("direct:update")  
        .end();  
}
```

The implementation of `StopRouteProcessor` is simple, as shown here:

```
public class StopRouteProcessor implements Processor {  
    private final String name;  
  
    public StopRouteProcessor(String name) {  
        this.name = name;  
    }  
  
    public void process(Exchange exchange) throws Exception  
{  
  
    exchange.getContext().getInflightRepository().remove(exchan  
ge, name);  
        exchange.getContext().stopRoute(name);  
    }  
}
```

This improves the readability of the route, because it's shorter and doesn't mix high-level routing logic with low-level implementation logic. Using `onCompletion`, the stopping logic has been separated from the original route.

You can use scopes to define `onCompletions` at different levels. Camel supports two scopes: context scope (high level) and route scope (low level). The preceding example uses context scope. If you want to use route scope, you have to define it within the route as follows:

```
from("file://target/inventory/manual?maxMessagesPerPoll=1")
    .onCompletion().process(new
StopRouteProcessor("manual")).end()
    .routeId("manual").noAutoStartup()
    .log("Doing manual update with file ${file:name}")
    .split(body()).tokenize("\n")
        .convertBodyTo(UpdateInventoryInput.class)
        .to("direct:update")
.end;
```

Notice the use of `.end()` to indicate where the `OnCompletion` route ends. You have to do this when using route scope so Camel knows which pieces belong to the *additional* route and which to the original route. This is the same principle as when you use `OnException` at route scope.

TIP `OnCompletion` also supports filtering using the `OnWhen` predicate so that you can trigger the additional route only if the predicate is `true`. In addition, `OnCompletion` can be configured to trigger only when the route completes successfully or when it fails using the `OnCompleteOnly` or `OnFailureOnly` options. For example, you can use `OnFailureOnly` to build a route that sends an email alert to support personnel when a route fails.

The book's source code contains this example in the `chapter15/startup` directory. You can try it using the following Maven goals:

```
mvn test -Dtest=ManualRouteWithOnCompletionTest
mvn test -Dtest=SpringManualRouteWithOnCompletionTest
```

We've covered how to stop a route at runtime using the `CamelContext` API. We'll now look at how to do this with the Control Bus EIP, which makes things simpler.

15.2.2 USING THE CONTROL BUS EIP TO START AND STOP ROUTES AT RUNTIME

Another way of controlling routes is using the Control Bus EIP. Yes, it's an enterprise integration pattern! Camel implements

this via the ControlBus component, instead of using a method in the DSL. By sending a message to a ControlBus endpoint, you can start, stop, resume, or suspend a route.

TIP Control Bus is also discussed in chapter 16, in section 16.4.3.

Looking back to the previous example in [listing 15.3](#), you can greatly simplify this with the Control Bus EIP:

```
from("file://target/inventory/manual?maxMessagesPerPoll=1")
    .routeId("manual").noAutoStartup()
    .log("Doing manual update with file ${file:name}")
    .split(body()).tokenize("\n")
        .convertBodyTo(UpdateInventoryInput.class)
        .to("direct:update")
    .end()
    .to("controlbus:route?
routeId=manual&action=stop&async=true");
```

The ControlBus component has a few options that you’re setting up here. The `routeId` refers to the route you’ll be controlling; `action` is `stop` because you want to stop the route. Finally, you set the `async` flag to make this action nonblocking—you don’t need a result set on the current exchange. Also notice that you don’t need to set up any executors this time or even manage the inflight repository of exchanges—it’s all handled by the Control Bus.

The book’s source code contains this example in the `chapter15/controlbus` directory. You can try it using the following Maven goal:

```
mvn test -Dtest=ControlBusTest
```

We’ve now covered how to stop a route at runtime using the Control Bus EIP. Let’s look at another feature called `RoutePolicy` that you can use to control the lifecycle of routes at runtime.

15.2.3 USING ROUTEPOLICY TO START AND

STOP ROUTES AT RUNTIME

RoutePolicy is a policy that can control routes at runtime. For example, RoutePolicy can control whether a route should be active. But you aren't limited to such scenarios; you can implement any kind of logic you want.

The org.apache.camel.spi.RoutePolicy interface defines several callback methods that Camel will automatically invoke at runtime:

```
public interface RoutePolicy {  
    void onInit(Route route);  
    void onRemove(Route route);  
    void onStart(Route route);  
    void onStop(Route route);  
    void onSuspend(Route route);  
    void onResume(Route route);  
    void onExchangeBegin(Route route, Exchange exchange);  
    void onExchangeDone(Route route, Exchange exchange);  
}
```

The idea is that you implement this interface, and Camel will invoke the callbacks when a new message arrives into a route, when a route has started or stopped, and so forth. You're free to implement whatever logic you want in these callbacks. For convenience, Camel provides the org.apache.camel.support.RoutePolicySupport class, which you can use as a base class to extend when implementing your custom policies.

Let's build a simple example using RoutePolicy to demonstrate how to flip between two routes, so only one route is active at any time. [Figure 15.5](#) shows this principle.

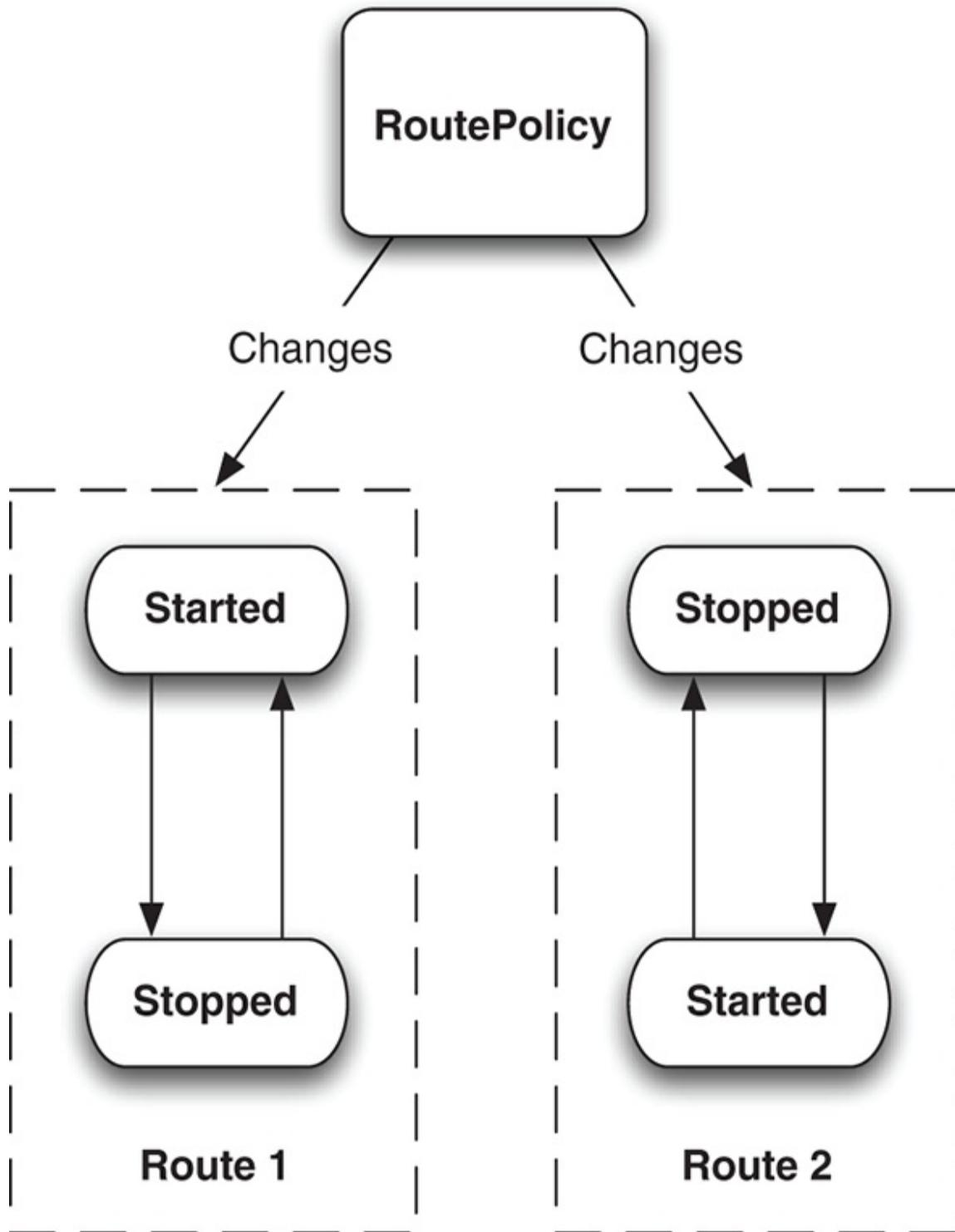


Figure 15.5 **RoutePolicy** changes the active state between the two routes so only one route is active at any time.

As you can see, **RoutePolicy** is being used to control the two routes, starting and stopping them so only one is active at a time.

The following listing shows how this can be implemented.

Listing 15.4 A RoutePolicy that flips two routes being active at runtime

```
public class FlipRoutePolicy extends RoutePolicySupport {  
    private final String name1;  
    private final String name2;
```

```
    public FlipRoutePolicy(String name1, String name2)  
    {  
        1
```

1

Identifies routes to flip

```
        this.name1 = name1;  
        this.name2 = name2;  
    }  
  
    @Override  
    public void onExchangeDone(Route route, Exchange  
exchange) {  
        String stop = route.getId().equals(name1) ? name1 :  
name2;  
        String start = route.getId().equals(name1) ? name2 :  
name1;  
        CamelContext context = exchange.getContext();  
        try {  
  
exchange.getContext().getInflightRepository().remove(exchan  
ge);  
        context.stopRoute(stop);  
        2
```

2

Flips the two routes

```
        context.startRoute(start);  
    } catch (Exception e) {  
        getExceptionHandler().handleException(e);  
    }  
}
```

In the constructor, you identify the names of the two routes to flip ❶. As you extend the `RoutePolicySupport` class, you override only the `onExchangeDone` method, as the flipping logic should be invoked when the route is done. You then compute which of the two routes to stop and start with the help of the `route` parameter, which denotes the current active route. Having computed that, you use `camelContext` to flip the routes ❷. If an exception is thrown, you let `ExceptionHandler` take care of it, which by default logs the exception.

To use `FlipRoutePolicy`, you must assign it to the two routes. In the Java DSL, this is done using the `RoutePolicy` method, as shown in the following `RouteBuilder`:

```
public void configure() throws Exception {
    RoutePolicy policy = new FlipRoutePolicy("foo", "bar");

    from("timer:foo")
        .routeId("foo").routePolicy(policy)
        .setBody().constant("Foo message")
        .to("log:foo").to("mock:foo");

    from("timer:bar")
        .routeId("bar").routePolicy(policy).noAutoStartup()
        .setBody().constant("Bar message")
        .to("log:bar").to("mock:bar");
}
```

If you're using XML DSL, you can use `RoutePolicy`, as shown here:

```
<bean id="flipPolicy"
class="camelaction.FlipRoutePolicy">
    <constructor-arg index="0" value="foo"/>
    <constructor-arg index="1" value="bar"/>
</bean>

<camelContext
xmlns="http://camel.apache.org/schema/spring">
    <route id="foo" routePolicyRef="flipPolicy">
        <from uri="timer:foo"/>
        <setBody><constant>Foo message</constant></setBody>
        <to uri="log:foo"/>
        <to uri="mock:foo"/>
    
```

```
</route>
<route id="bar" routePolicyRef="flipPolicy"
autoStartup="false">
    <from uri="timer:bar"/>
    <setBody><constant>Bar message</constant></setBody>
    <to uri="log:bar"/>
    <to uri="mock:bar"/>
</route>
</camelContext>
```

As you can see, you use the `routePolicyRef` attribute on the `<route>` tag to reference the `flipPolicy` bean defined in the top of the XML file.

The book's source code contains this example in the `chapter15/routepolicy` directory. You can try it using the following Maven goals:

```
mvn test -Dtest=FlipRoutePolicyJavaDSLTest
mvn test -Dtest=FlipRoutePolicySpringXMLTest
```

When running either of the examples, you should see the two routes being logged interchangeably (`foo` and `bar`):

```
INFO  foo - Exchange[BodyType:String, Body:Foo message]
INFO  bar - Exchange[BodyType:String, Body:Bar message]
INFO  foo - Exchange[BodyType:String, Body:Foo message]
INFO  bar - Exchange[BodyType:String, Body:Bar message]
INFO  foo - Exchange[BodyType:String, Body:Foo message]
INFO  bar - Exchange[BodyType:String, Body:Bar message]
```

We've now covered both starting and controlling routes at runtime. It's time to learn about shutting down Camel, which is more complex than it sounds.

15.3 *Shutting down Camel*

The new inventory application at Rider Auto Parts is scheduled to be in production at the end of the month. You're on the team to ensure its success and help run the final tests before it's handed over to production. These tests also cover reliably shutting down the application.

Shutting down the Camel application is complex because numerous in-flight messages could be being processed. Shutting down while messages are in flight may harm your business, because those messages could be lost. The goal of shutting down a Camel application reliably is to shut it down when it's *quiet*—when there are no in-flight messages. All you have to do is find this quiet moment.

This is hard to do, because while you wait for the current messages to complete, the application may take in new messages. You have to stop taking in new messages while the current messages are given time to complete. This process is known as *graceful shutdown*, which means shutting down in a reliable and controlled manner.

15.3.1 GRACEFUL SHUTDOWN

When `CamelContext` is being stopped, which happens when its `stop()` method is invoked, it uses a strategy to shut down. This strategy is defined in the `ShutdownStrategy` interface. The default implementation of this `shutdownStrategy` interface uses the graceful shutdown technique.

For example, when you stop the Rider Auto Parts example, you'll see these log lines:

```
SpringCamelContext      - Apache Camel 2.20.1
(CamelContext: camel-1)           is shutting down
DefaultShutdownStrategy - Starting to graceful shutdown 3
routes                      (timeout 10 seconds)
DefaultShutdownStrategy - Route: webservice shutdown
complete, was                consuming from:
                               cxf://bean:inventoryEndpoint
DefaultShutdownStrategy - Route: file shutdown complete,
was consuming                 from:
                               file://target/inventory/updates
DefaultShutdownStrategy - Route: update shutdown complete,
was consuming                from: direct://update
```

```
DefaultShutdownStrategy - Graceful shutdown of 3 routes
completed in          0 seconds
SpringCamelContext     - Apache Camel 2.20.1
(CamelContext: camel-1)
SpringCamelContext     uptime 0.684 seconds
(CamelContext: camel-1)
                                         - Apache Camel 2.20.1
                                         is shutdown in 0.025 seconds
```

This tells you a few things. You can see that the graceful shutdown is using a 10-second time-out. This is the maximum time Camel allows for shutting down gracefully before it starts to shut down more aggressively by forcing routes to stop immediately. The default value is 300 seconds, but the log was a unit test, and `camelTestSupport` uses 10 seconds. You can configure this time-out yourself on `CamelContext`. For example, to use 20 seconds as the default time-out value, you can do the following:

```
camelContext.getShutdownStrategy().setTimeout(20);
```

Doing this in XML requires a bit more work, because you have to define a `<bean>` to set the time-out value:

```
<bean id="shutdown"
  class="org.apache.camel.impl.DefaultShutdownStrategy">
    <property name="timeout" value="20"/>
</bean>
```

Notice that the time-out value is in seconds.

Then Camel logs the progress of the routes as they shut down, one by one, according to the reverse order in which they were started.

At the end, Camel logs the completion of the graceful shutdown, which in this case was fast and completed in less than 1 second. Camel also logs whether there were any in-flight messages just before it stops completely.

If any in-flight exchanges are holding up the graceful shutdown, Camel prints a status about these every second, as

follows:

```
DefaultShutdownStrategy - Waiting as there are still 3
inflight and
pending exchanges to complete, timeout in 60 seconds
Inflights per route: [file = 2, update = 1]
```

This logging continues until the in-flight exchanges complete or the time-out is reached. A time-out indicates that Camel didn't complete the graceful shutdown, and it would log a warning showing the in-flight exchanges that didn't complete:

```
WARN DefaultShutdownStrategy - Timeout occurred during
graceful shutdown.
                                         Forcing the routes to be
                                         shutdown now.
                                         Notice: some resources may
                                         still be running
                                         as graceful shutdown did
                                         not complete
                                         successfully.
WARN DefaultShutdownStrategy - Interrupted while waiting
during graceful
                                         shutdown, will force
                                         shutdown now.
INFO DefaultShutdownStrategy - Route: file shutdown
complete, was
                                         consuming from:
file://target/inventory/updates
INFO DefaultShutdownStrategy - Route: update shutdown
complete, was
                                         consuming from:
direct://update
INFO DefaultShutdownStrategy - There are 2 inflight
exchanges:
    InflightExchange: [exchangeId=ID-ghost-33143-
1479685285199-0-10,
    fromRouteId=file, routeId=update, nodeId=to3,
elapsed=17, duration=3100]
    InflightExchange: [exchangeId=ID-ghost-33143-
1479685285199-0-4,
    fromRouteId=file, routeId=file, nodeId=split1,
elapsed=3100, duration=3100]
```

All this additional logging may not be desirable for all applications. Maybe it's expected that on shutdown, messages

are discarded. In such cases, you can disable the additional logging. To suppress the warning on time-out, you can do the following:

```
context.getShutdownStrategy().setSuppressLoggingOnTimeout(true);
```

To suppress the status messages about in-flight exchanges during graceful shutdown, you can do this:

```
context.getShutdownStrategy().setLogInflightExchangesOnTimeOut(false);
```

SHUTTING DOWN THE RIDER AUTO PARTS APPLICATION

At Rider Auto Parts, you're in the final testing of the application before it's handed over to production. One of the tests is based on processing a big inventory file, and you want to test what happens if you shut down Camel while it's working on the big file. Let's look at how Camel handles this.

At first, you see the usual logging about the shutdown in progress:

```
SpringCamelContext      - Apache Camel 2.20.1
(CamelContext: camel-1) is
                           shutting down
DefaultShutdownStrategy - Starting to graceful shutdown 3
routes
                           (timeout 60 seconds)
DefaultShutdownStrategy - Route: webservice shutdown
complete, was
                           consuming from:
cxf://bean:inventoryEndpoint
```

Then there's a log line indicating that Camel has noticed in-flight exchanges, which are from the big file, and also exchanges generated from lines of the big file. This is expected behavior:

DefaultShutdownStrategy - Waiting as there are still 3 inflight and pending exchanges to complete, timeout in 60 seconds.

Inflights per route: [file = 2,

```
update = 1]
```

The application logs the progress of the inventory update, which should happen for each line in the big file:

```
Inventory 58004 updated
```

Finally, you see the last log lines, which report the end of the shutdown:

```
DefaultShutdownStrategy - Route: file shutdown complete,  
was consuming  
                                from:  
file://target/inventory/updates  
DefaultShutdownStrategy - Route: update shutdown complete,  
was consuming  
                                from: direct://update  
DefaultShutdownStrategy - Graceful shutdown of 3 routes  
completed in 14  
                                seconds  
SpringCamelContext      - Apache Camel 2.20.1  
(CamelContext: camel-1)  
                                uptime 17.747 seconds  
SpringCamelContext      - Apache Camel 2.20.1  
(CamelContext: camel-1)  
                                is shutdown in 14.028 seconds
```

Notice that the routes are shut down in the reverse order in which they were started. This is handy because it maintains any order dependency they had during startup. Previous versions of Camel behaved differently than this, and you often had to defer shutdown of routes that were required longer than their startup order implied. For example, to defer shutdown of the `update` route, you could use the `ShutdownRoute` option, which was listed in table [15.1](#). All you have to do is add the option in the route, as shown in bold here:

```
from("direct:update")  
    .routeId("update").startupOrder(1)  
    .shutdownRoute(ShutdownRoute.Defer)  
    .to("bean:inventoryService?method=updateInventory");
```

The same route in XML DSL is as follows:

```
<route id="update" startupOrder="1" shutdownRoute="Defer">
```

```
<from uri="direct:update"/>
<to uri="bean:inventoryService?
method=updateInventory"/>
</route>
```

The book's source code contains this example in the chapter15/shutdown directory. You can try it using the following Maven goals:

```
mvn test -Dtest=GracefulShutdownBigFileTest
mvn test -Dtest=GracefulShutdownBigFileXmlTest
mvn test -Dtest=GracefulShutdownBigFileDeferTest
mvn test -Dtest=GracefulShutdownBigFileDeferXmlTest
```

As you've seen, Camel gives end users full control over how to configure their routes to ensure a proper shutdown. That said, most of the time you don't need to worry about manually ordering your routes. It's nice to have the flexibility, though, just in case you need to work around an issue.

ABOUT STOPPING AND SHUTTING DOWN

Camel uses the graceful shutdown mechanism when it stops or shuts down routes, so [listing 15.3](#) will stop the route in a graceful manner. As a result, you can reliably stop routes at runtime without the risk of losing in-flight messages.

The difference between using the `stopRoute` and `shutdownRoute` methods is that the latter also unregisters the route from management (JMX). Use `stopRoute` when you want to be able to start the route again; use `shutdownRoute` only if the route should be permanently removed.

That's all there is to shutting down Camel. It's now time to review some of the possible deployment strategies.

15.4 Deploying Camel

Camel is described as a lightweight and embeddable integration framework. It supports more deployment strategies and flexibility than traditional ESBs and application servers do.

Camel can be used in a wide range of runtime environments, from standalone Java applications to web containers to the cloud.

This section presents five deployment strategies that are possible with Camel and their strengths and weaknesses:

- Embedding Camel in a Java application
- Running Camel in a web environment such as Apache Tomcat
- Running Camel inside WildFly
- Running Camel in an OSGi container such as Apache Karaf
- Running Camel in a container that supports CDI, such as Apache Karaf or WildFly

It's also possible to deploy Camel using Spring Boot and WildFly Swarm. We cover those two deployment options in chapter 7, where we discuss microservices.

15.4.1 EMBEDDED IN A JAVA APPLICATION

Embedding Camel in a Java application is appealing if you need to communicate with the outside world. By bringing Camel into your application, you can benefit from all the transports, routes, EIPs, and so on that Camel offers. [Figure 15.6](#) shows Camel embedded in a standard Java application.

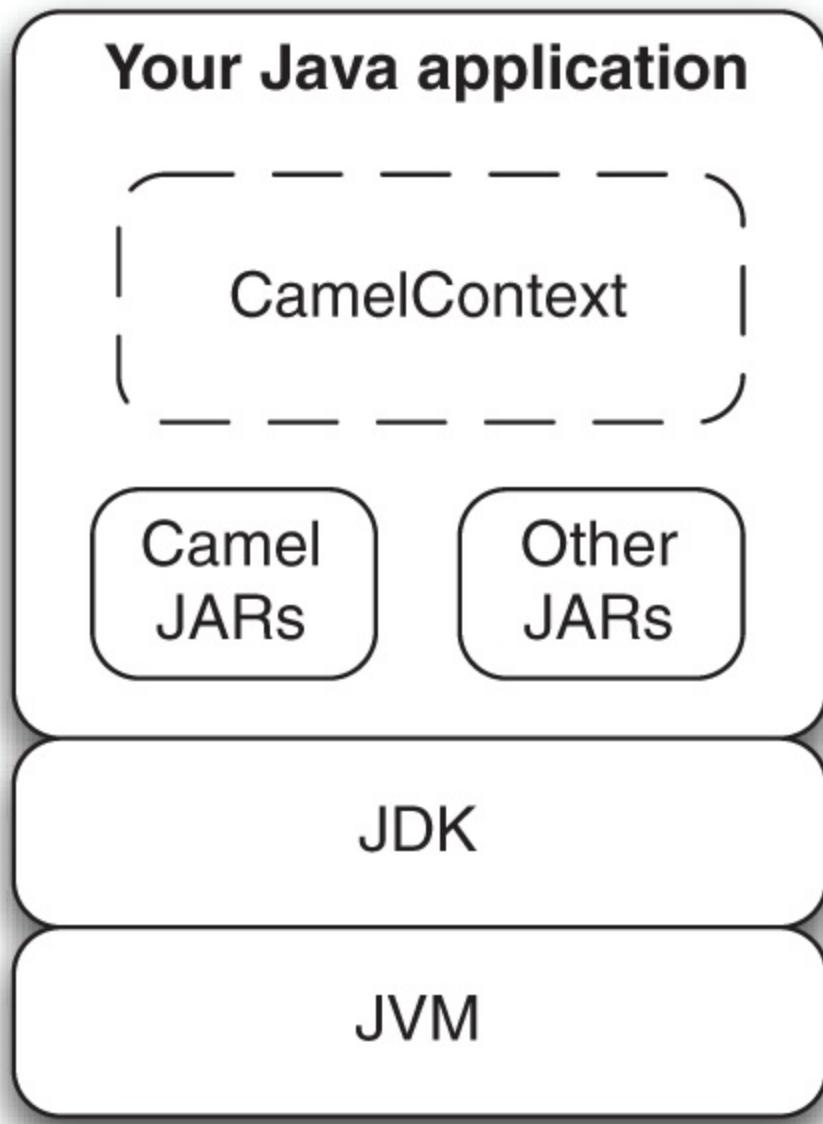


Figure 15.6 Camel embedded in a standalone Java application

In the embedded mode, you must add to the classpath all the necessary Camel and third-party JARs needed by the underlying transports. Because Camel is built with Maven, you can use Maven for your own project and benefit from its dependency-management system. We discuss building Camel projects with Maven in chapter 8.

Bootstrapping Camel for your code is easy. In fact, you did that in your first ride on the Camel in chapter 1. All you need to do is

create `camelContext` and start it, as shown in the following listing.

Listing 15.5 Bootstrapping Camel in your Java application

```
public class FileCopierWithCamel {  
    public static void main(String args...) throws  
Exception {  
        CamelContext context = new DefaultCamelContext();  
        context.addRoutes(new RouteBuilder() {  
            public void configure() {  
  
from("file:data/inbox").to("file:data/outbox");  
            }  
        });  
        context.start();  
        Thread.sleep(10000);  
        context.stop();  
    }  
}
```

It's important to keep a reference to `camelContext` for the lifetime of your application, because you'll need to perform a shutdown of Camel. As you may have noticed in [listing 15.5](#), the code execution continues after starting `camelContext`. To avoid shutting down your application immediately, the listing includes code to sleep for 10 seconds. In your application, you'll need to use a different means to let your application continue and shut down only when requested to do so.

Let's look at the Rider Auto Parts application illustrated in [figure 15.1](#) and embed it in a Java application.

EMBEDDING THE RIDER AUTO PARTS EXAMPLE IN A JAVA APPLICATION

The Rider Auto Parts Camel application can be run as a standalone Java application. Because the application is using Spring XML files to set up Camel, you need to start only Spring to start the application.

Starting Spring from a main class can be done as follows:

```
public class InventoryMain {  
    public static void main(String[] args) throws Exception
```

```

{
    String filename = "META-INF/spring/camel-
context.xml";
    AbstractXmlApplicationContext spring =
        new
ClassPathXmlApplicationContext(filename);
    spring.start();
    Thread.sleep(10000);
    spring.stop();
    spring.destroy();
}
}

```

To start Spring, you create `ApplicationContext`, which in the preceding example means loading the Spring XML file from the classpath. This code also reveals the problem of having the main method wait until you terminate the application. The preceding code uses `Thread.sleep` to wait 10 seconds before terminating the application.

To remedy this, Camel provides a `Main` class that you can use instead of writing your own class. You can change the previous `InventoryMain` class to use this `Main` class as follows:

```

import org.apache.camel.spring.Main;

public class InventoryMain {
    public static void main(String[] args) throws Exception
{
    Main main = new Main();
    main.setApplicationContextUri("META-
INF/spring/camel-context.xml");
    main.run();
}
}

```

This approach also solves the issue of handling the lifecycle of the application. Camel will shut down gracefully when the JVM is being terminated, such as when the Ctrl-C key combination is pressed. You can also stop the Camel application by invoking the `stop` method on the `main` instance.

The book's source code contains this example in the `chapter15/standalone` directory. You can try it using the

following Maven goal:

```
mvn compile exec:java
```

TIP You may not need to write your own main class. For example, the `org.apache.camel.main.Main`, `org.apache.camel.spring.Main`, and `org.apache.camel.cdi.Main` classes can be used directly. They have parameters to dictate which `RouteBuilder` class or Spring XML file should be loaded. By default, the `org.apache.camel.spring.Main` class loads all XML files from the classpath in the `META-INF/spring` location, so by dropping your Spring XML file in there, you don't even have to pass any arguments to the `Main` class. You can start it directly.

Table 15.2 summarizes the pros and cons of embedding Camel in a standalone Java application.

Table 15.2 Pros and cons of embedding Camel in a standalone Java application

Pros	Cons
<ul style="list-style-type: none">• Gives flexibility to deploy just what's needed• Allows you to embed Camel in any standard Java application	<ul style="list-style-type: none">• Requires deploying all needed JARs• Requires manually managing Camel's lifecycle (starting and stopping) on your own

You now know how to make your standalone application use Camel. Let's look at what you can do for your web applications.

15.4.2 EMBEDDED IN A WEB APPLICATION

Embedding Camel in a web application brings the same benefits as those mentioned in section 15.4.1. Camel provides all you need to connect to your favorite web container. If you work in an

organization, you may be able to use existing infrastructure for deploying your Camel applications. Deploying Camel in such a well-known environment gives you immediate support for installing, managing, and monitoring your applications.

When Camel is embedded in a web application, as shown in [figure 15.7](#), you need to make sure all JARs are packaged in the WAR file. If you use Maven, this is done automatically.

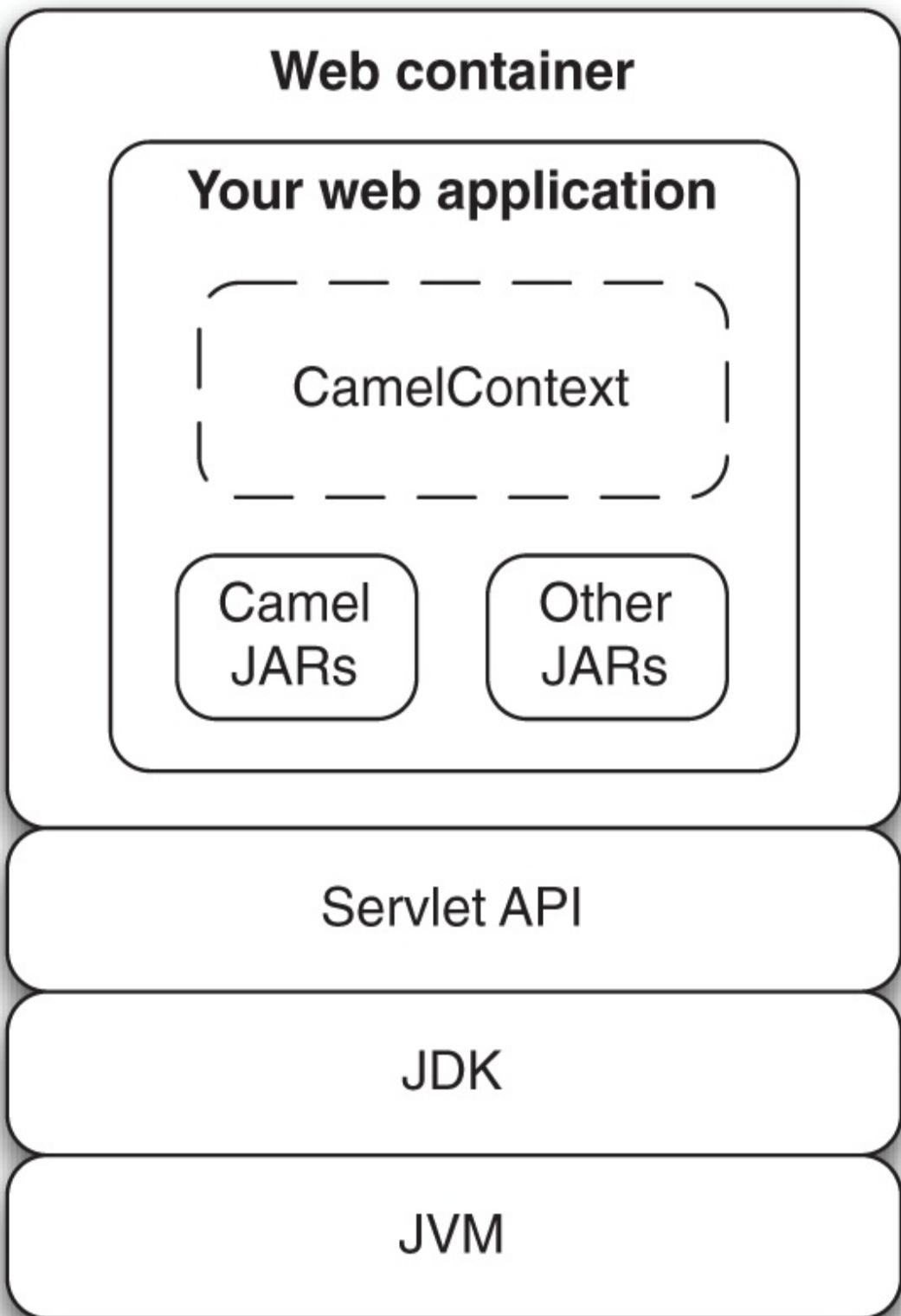


Figure 15.7 Camel embedded in a web application

The Camel instance embedded in your web application is bootstrapped by Spring. Using Spring, which is such a ubiquitous framework, you let end users use well-known approaches for deployment. This also conveniently ties Camel's lifecycle with Spring's lifecycle management and ensures that the Camel instance is properly started and stopped in the web container.

The following code demonstrates that you need only a standard Spring context listener in the web.xml file to bootstrap Spring and thereby Camel:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
          xsi:schemaLocation="
              http://java.sun.com/xml/ns/javaee
              http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
          version="3.0">
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
</web-app>
```

This context listener also takes care of properly shutting down Camel when the web application is stopped. Spring will, by default, load the Spring XML file from the WEB-INF folder using the name applicationContext.xml. In this file, you can embed Camel, as shown in [listing 15.6](#).

Specifying the location of your Spring XML file

If you want to use another name for your Spring XML file, you need to add a context parameter that specifies the

filename as follows:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/camel-context.xml</param-
value>
</context-param>
```

Listing 15.6 The Spring applicationContext.xml file with Camel embedded

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-
beans.xsd
           http://camel.apache.org/schema/spring
           http://camel.apache.org/schema/spring/camel-
spring.xsd">
    <import resource="camel-cxf.xml"/> ①
```

1

Imports CXF from another XML file

```
<bean id="inventoryService"
      class="camelinaction.InventoryService"/>
    <bean id="inventoryRoute"
      class="camelinaction.InventoryRoute"/>
    <camelContext
      xmlns="http://camel.apache.org/schema/spring">
        <routeBuilder ref="inventoryRoute"/>
    </camelContext>
</beans>
```

Listing 15.6 is a regular Spring XML file in which you can use the `<import>` tag to import other XML files. For example, you do this by having CXF defined in the `camel-cxf.xml` file ①. Camel itself

is embedded using the <camelContext> tag.

The book's source code contains this example in the chapter15/war directory. You can try it using the following Maven goal:

```
mvn jetty:run
```

If you run this Maven goal, a Jetty plugin is used to quickly boot up a Jetty web container running the web application. To use the Jetty plugin in your projects, you must remember to add it to your pom.xml file in the <build><plugins> section:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>maven-jetty-plugin</artifactId>
  <version>9.2.21.v20170120</version>
</plugin>
```

If you run this goal, you should notice in the console that Jetty has been started:

```
2016-11-27 17:05:44,884 [main] INFO
SpringCamelContext
- Apache Camel 2.20.1 (CamelContext: camel-1) started in
0.788 seconds
...
[INFO] Started ServerConnector@2d2f09a4{HTTP/1.1}
{0.0.0.0:8080}
[INFO] Started @34700ms
[INFO] Started Jetty Server
```

Let's look at running Camel as a web application in Apache Tomcat.

DEPLOYING TO APACHE TOMCAT

To package the application as a WAR file, you can run the mvn package command, which creates the WAR file in the target directory. Yes, it's that easy with Maven.

You want to use the hot deployment of Apache Tomcat, so you must first start it. Here's how to start it on a Linux system using the bin/startup.sh script:

```
[janstey@ghost apache-tomcat-8.5.23]$ bin/startup.sh
Using CATALINA_BASE:      /home/janstey/kits/apache-tomcat-
8.5.23
Using CATALINA_HOME:      /home/janstey/kits/apache-tomcat-
8.5.23
Using CATALINA_TMPDIR:   /home/janstey/kits/apache-tomcat-
8.5.23/temp
Using JRE_HOME:           /usr/java/jdk1.8.0_91/
Using CLASSPATH:          /home/janstey/kits/apache-tomcat-
8.5.23/bin/bootstrap.jar:/home/janstey/kits/apache-tomcat-
8.5.23/bin/tomcat-juli.jar
Tomcat started.
```

This starts Tomcat in the background, so you need to tail the log file to see what happens:

```
[janstey@ghost apache-tomcat-8.5.23]$ tail -f
logs/catalina.out
27-Nov-2016 17:41:38.856 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory
Deployment of web
application directory /home/janstey/kits/apache-tomcat-
8.5.8/webapps/examples has finished in 175 ms
...
27-Nov-2016 17:41:38.898 INFO [main]
org.apache.coyote.AbstractProtocol.start Starting
ProtocolHandler
[http-nio-8080]
27-Nov-2016 17:41:38.902 INFO [main]
org.apache.coyote.AbstractProtocol.start Starting
ProtocolHandler
[ajp-nio-8009]
27-Nov-2016 17:41:38.902 INFO [main]
org.apache.catalina.startup.Catalina.start Server startup
in 461 ms
```

To deploy the application, you need to copy the WAR file to the Apache Tomcat webapps directory:

```
cp target/riderautoparts-war-2.0.0.war ~/kits/apache-
tomcat-8.5.23/webapps/
```

Apache Tomcat should show the application being started in the log file. You should see the familiar logging of Camel being started:

```
2016-11-27 17:45:22,507 [ost-startStop-2] INFO
SpringCamelContext
- Total 3 routes, of which 3 are started.
2016-11-27 17:45:22,509 [ost-startStop-2] INFO
SpringCamelContext
- Apache Camel 2.20.1 (CamelContext: camel-1) started in
1.037 seconds
2016-11-27 17:45:22,513 [ost-startStop-2] INFO
ContextLoader
- Root WebApplicationContext: initialization completed in
2604 ms
```

Now you need to test that the deployed application runs as expected by sending a web service request using SoapUI, as shown in [figure 15.8](#). Doing this requires you to know the URL to the WSDL the web service runs at, which is <http://localhost:9000/inventory?wsdl>.

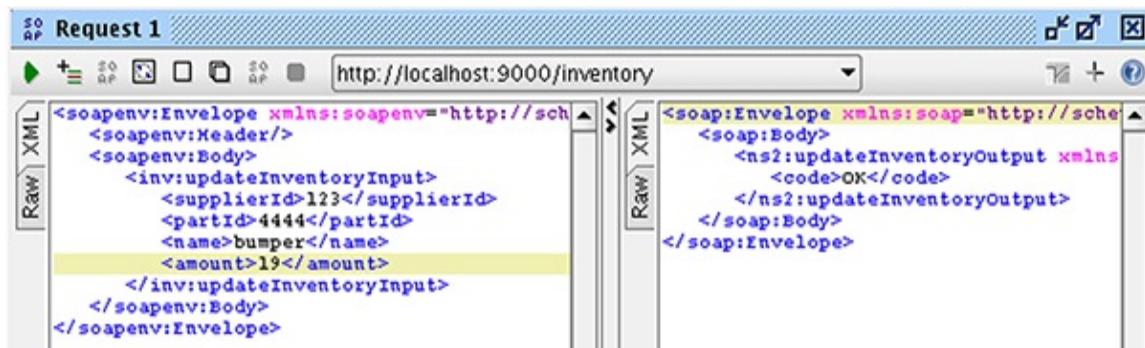


Figure 15.8 Using SoapUI to test the web service from the deployed application in Apache Tomcat

The web service returns `ok` as its reply, and you can also see from the log file that the application works as expected, outputting the inventory being updated:

```
Inventory 4444 updated
```

Another great benefit of this deployment model is that you can tap the servlet container directly for HTTP endpoints. In a standalone Java deployment scenario, you have to rely on the Jetty transport, but in the web deployment scenario, the container already has its socket management, thread pools, tuning, and monitoring facilities. Camel can use these container-provided features if you use the servlet transport for your

inbound HTTP endpoints.

In the previously deployed application, you let Apache CXF rely on the Jetty transport. Let's change this to use the existing servlet transports provided by Apache Tomcat.

USING APACHE TOMCAT FOR HTTP INBOUND ENDPOINTS

When using Camel in an existing servlet container, such as Apache Tomcat, you may have to adjust Camel components in your application to tap into the servlet container. In the Rider Auto Parts application, it's the CXF component you must adjust. First, you have to add `cxfServlet` to the `web.xml` file, as shown in the following listing.

Listing 15.7 The `web.xml` file with `cxfServlet` to tap into Apache Tomcat

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
          xsi:schemaLocation="
              http://java.sun.com/xml/ns/javaee
              http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
          version="3.0">
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>CXFServlet</servlet-name>
        <servlet-class>
            org.apache.cxf.transport.servlet.CXFServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Maven users need to adjust the pom.xml file to depend on the HTTP transport instead of Jetty, as follows:

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http</artifactId>
</dependency>
```

Next, you must adjust the camel-cxf.xml file, as shown in the following listing.

Listing 15.8 Setting up the Camel CXF component to tap into Apache Tomcat

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
       xmlns:cxf="http://camel.apache.org/schema/cxf"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-
beans.xsd
           http://camel.apache.org/schema/cxf
           http://camel.apache.org/schema/cxf/camel-cxf.xsd">
    <import resource="classpath:META-INF/cxf/cxf.xml"/>
    <import resource=
        "classpath:META-INF/cxf/cxf-servlet.xml"/> 1

```

1

Required import when using servlet

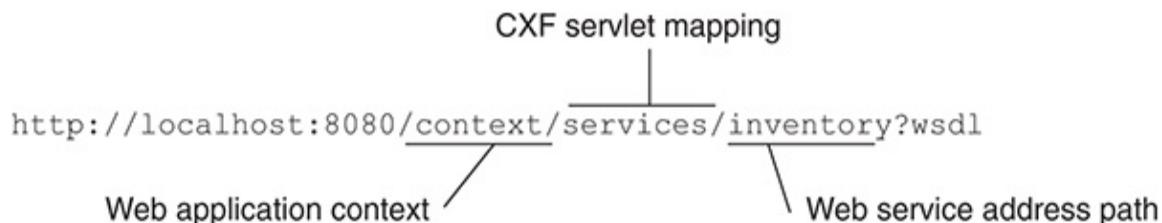
```
<cxf:cxfEndpoint id="inventoryEndpoint"
                  address="/inventory" 2
```

2

Web service endpoint address

```
serviceClass="camelinaction.inventory.InventoryEndpoint"/>
</beans>
```

To use Apache CXF in a servlet container, you have to import the `cxn-servlet.xml` resource ❶. This exposes the web service via the servlet container, which means the endpoint address has to be adjusted to a relative context path ❷.



In the previous example, the web service was available at `http://localhost:9000/inventory?wsdl`. Using Apache Tomcat, the web service is now exposed at a different address. Notice that the TCP port is 8080, which is the default Apache Tomcat setting.

The book's source code contains this example in the `chapter15/war-servlet` directory. You can package the application using `mvn package` and then copy the `riderautoparts-war-servlet-2.0.0.war` file to the `webapps` directory of Apache Tomcat to hot-deploy the application. Then the web service should be available at this address: `http://localhost:8080/riderautoparts-war-servlet-2.0.0/services/inventory?wsdl`.

NOTE Camel also provides a lightweight alternative to using CXF in the servlet container. The `Servlet` component allows you to consume HTTP requests coming into the servlet container in much the same way as you saw here with CXF. You can find more information on the Apache Camel website at <http://camel.apache.org/servlet.html>.

Table 15.3 lists the pros and cons of the web application deployment model of Camel.

Table 15.3 Pros and cons of embedding Camel in a web application

Pros	Cons

- | | |
|---|--|
| <ul style="list-style-type: none"> • Taps into the servlet container • Lets the container manage the Camel lifecycle • Benefits the management and monitoring capabilities of the servlet container • Provides familiar runtime platform for operations | <ul style="list-style-type: none"> • Can create annoying classloading issues on some web containers |
|---|--|

Embedding Camel in a web application is a popular, proven, and powerful way to deploy Camel. Another choice for running Camel applications is using an application server such as WildFly (which is the new name for JBoss Application Server).

15.4.3 EMBEDDED IN WILDFLY

A common way of deploying Camel applications in WildFly is using the web deployment model discussed in the previous section. But WildFly has a slightly different classloading mechanism, so you need to take care when loading things such as type converters. If you’re using the camel-bindy component, you’ll even have to use a special Camel JBoss component to adapt to this classloading. This component isn’t provided out of the box with the Apache Camel distribution because of license implications with WildFly’s LGPL license. This component is hosted at Camel Extra (<https://github.com/camel-extra/camel-extra>), which is a project site for additional Camel components that can’t be shipped from Apache.

For most cases, you won’t need this component, so we won’t dwell on that here. You need to make just one modification to the war-servlet example in the previous section to get it to deploy on WildFly. For WildFly deployments, you can’t rely on class resolution by package name. For our single type converter, you

need to tell Camel the fully qualified name of the type-converter class to load. You make a change to src/main/resources/META-INF/services/org/apache/camel/TypeConverter as follows:

```
-camelinaction  
+camelinaction.InventoryConverter
```

That's all there is to it!

To deploy the application to WildFly, you start it and copy the WAR file into the standalone/deployments directory. For example, on our laptop, WildFly 11.0.0.Final is started as follows:

```
[janstey@ghost wildfly-11.0.0.Final]$ ./bin/standalone.sh
```

After a couple of seconds, WildFly is ready, and this is logged to the console:

```
00:57:47,693 INFO [org.jboss.as] (Controller Boot Thread)  
WFLYSRV0025: WildFly Full 11.0.0.Final (WildFly Core  
3.0.8.Final) started in 2437ms - Started 292 of 553  
services (347 services are lazy, passive or on-demand)
```

Then the WAR file is copied:

```
[janstey@ghost war-wildfly]$ cp target/riderautoparts-war-  
wildfly-2.0.0.war ~/kits/wildfly-  
11.0.0.Final/standalone/deployments/
```

You can then keep an eye on the WildFly console as it outputs the progress of the deployment. WildFly uses an embedded instance of Undertow as the servlet container, which is configured to have a similar location as before with Tomcat:

```
http://localhost:8080/riderautoparts-war-wildfly-  
2.0.0/services/inventory?wsdl
```

Table 15.4 lists the pros and cons of deploying Camel as a WAR in WildFly.

Table 15.4 Pros and cons of embedding Camel as a WAR in WildFly

Pros	Cons

<ul style="list-style-type: none"> • Taps into the WildFly container. • Allows your application to use the facilities provided by the Java EE application server. • Lets the WildFly container manage Camel's lifecycle. • Benefits the management and monitoring capabilities of the application server. • Provides a familiar runtime platform for operations. 	<ul style="list-style-type: none"> • Sometimes requires a special Camel component to remedy classloading issues. • WARs have to package many dependencies, which leads to a "fat" deployment artifact.
---	--

Deploying Camel as a web application in WildFly is a fairly easy solution. All you have to remember is to use the special Camel JBoss component to let the class loading work for the camel-bindy component and to use the fully qualified class name when loading type converters.

THE WILDFLY-CAMEL SUBSYSTEM

There's another way to deploy Camel applications to WildFly that provides a much tighter integration between Camel and WildFly, and that is using a project called the WildFly-Camel Subsystem.¹ Why use this project? Well, you may not have noticed, but standard WAR-style deployment results in a pretty large WAR for many Camel applications. With WildFly-Camel, you don't have to include any of the Camel libraries in your WAR. Also, the versions of various third-party libraries that come with a Camel component may conflict with what's included in the WildFly container. WildFly-Camel takes care of this dependency management for you so you can focus on the Camel

application.

¹ Check out the project source at <https://github.com/wildfly-extras/wildfly-camel> and docs at <https://wildflyext.gitbooks.io/wildfly-camel/content>.

You can even write Camel routes directly in WildFly XML configuration, like this:

```
<server xmlns="urn:jboss:domain:4.2">
  <profile>
    ...
      <subsystem xmlns="urn:jboss:domain:camel:1.0">
        <camelContext id="system-context-1">
          <![CDATA[
            <route id="file">
              <from uri="file:///target/inventory/updates"/>
              <log message="Received order #{body}"/>
            </route>
          ]]>
        </camelContext>
      </subsystem>
    ...
  </profile>
</server>
```

To try it, you first need to install WildFly-Camel, which is a separate project from WildFly. You can find instructions on how to install WildFly-Camel in the project's documentation at <http://wildfly-extras.github.io/wildfly-camel/>. At the time of writing, the latest version was 5.0.0, so we used that.

After installing WildFly-Camel, you can boot up WildFly with Camel support by running this:

```
./bin/standalone.sh -c standalone-full-camel.xml
```

If standalone/configuration/standalone-full-camel.xml contained a `<camelContext>` element as shown previously, it'd be started with the container, and you'd see something like this in the logs:

```
17:35:49,523 INFO [org.wildfly.extension.camel] (MSC service thread 1-8)
  Camel context starting: system-context-1
```

```
17:35:49,885 INFO
[org.apache.camel.spring.SpringCamelContext] (MSC
    service thread 1-8) Route: file started and consuming
from: file://target/
    inventory/updates
17:35:49,885 INFO
[org.apache.camel.spring.SpringCamelContext] (MSC
    service thread 1-8) Total 1 routes, of which 1 are
started.
17:35:49,886 INFO
[org.apache.camel.spring.SpringCamelContext] (MSC
    service thread 1-8) Apache Camel 2.20.1 (CamelContext:
system-context-1)
    started in 0.465 seconds
```

Putting all your routes into the main WildFly configuration isn't exactly a scalable solution. You need to be able to deploy applications separately from this file. One other way is by defining your CamelContext in an XML file under META-INF and named `*-camel-context.xml`. For example, the previous WAR example had `CamelContext` defined in `src/main/webapp/WEB-INF/applicationContext.xml`. WildFly-Camel will search in META-INF, so you need to move and rename that file to `src/main/resources/META-INF/inventory-camel-context.xml`. The difference in the WAR size is apparent right away:

```
wildfly-war/target/riderautoparts-war-wildfly-2.0.0.war 15M
wildfly-camel-war/target/riderautoparts-wildfly-camel-war-
2.0.0.war 16K
```

To ensure that you end up with this skinny WAR file, you must use the provided scope for all Camel dependencies in your POM. For our example, you have dependencies like this:

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

```

<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-cxf</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http</artifactId>
    <version>3.2.1</version>
    <scope>provided</scope>
</dependency>

```

To deploy this example to a locally running WildFly instance, go to the chapter15/wildfly-camel-war directory of the book's source code and run this:

```
mvn clean install -Pdeploy
```

That builds a skinny WAR with our inventory-camel-context.xml and supporting Java files and then deploys to WildFly.

TIP Back in chapter 7, we also used the WildFly-Camel Subsystem with WildFly Swarm.

You can also avoid XML configuration altogether by annotating your Java Camel routes with CDI annotations. This is covered

further in section 15.6.

Table 15.5 lists the pros and cons of deploying Camel using WildFly-Camel.

Table 15.5 Pros and cons of deploying Camel using WildFly-Camel

Pros	Cons
<ul style="list-style-type: none">• Taps into the WildFly container• Allows your application to use the facilities provided by the Java EE application server• Lets the WildFly container manage Camel's lifecycle• Benefits the management and monitoring capabilities of the application server• Provides a familiar runtime platform for operations• Deployments are much smaller in size compared to regular WAR style deployment	<ul style="list-style-type: none">• Not all Camel components available

The last strategy we'll cover is in a totally different ballpark: using OSGi. OSGi brings the promise of modularity to the extreme.

15.5 Camel and OSGi

OSGi is a layered module system for the Java platform that offers a complete dynamic component model. It's a truly dynamic environment in which components can come and go without requiring a reboot (hot deployment). Apache Camel is OSGi ready, in the sense that all the Camel JAR files are OSGi compliant and are deployable in OSGi containers.

This section shows you how to prepare and deploy the Rider Auto Parts application in the Apache Karaf OSGi runtime. Karaf provides functionality on top of the OSGi container, such as hot deployment, provisioning, local and remote shells, and many other goodies. You can choose between Apache Felix or Eclipse Equinox for the OSGi container. In addition, all of what we're about to discuss also applies to containers that build on top of Karaf, such as JBoss Fuse and Apache ServiceMix.

The example presented here is included with the book's source code in the chapter15/osgi directory.

NOTE This book doesn't go deep into the details of OSGi, which is a complex topic. The basics are covered on Wikipedia (<http://en.wikipedia.org/wiki/OSGi>), and if you're interested in more information, we highly recommend *OSGi in Action* by Richard S. Hall et al. (Manning, 2011). For more information on the Apache Karaf OSGi runtime, see the Karaf website: <http://karaf.apache.org>.

The first thing you need to do with the Rider Auto Parts application is make it OSGi compliant (packaged as an OSGi bundle). This involves setting up Maven to help prepare the packaged JAR file so it includes OSGi metadata in the `MANIFEST.MF` entry.

15.5.1 SETTING UP MAVEN TO GENERATE AN OSGI BUNDLE

In the `pom.xml` file, you have to set the packaging element to `bundle`, which means the JAR file will be packaged as an OSGi bundle:

```
<packaging>bundle</packaging>
```

To generate the `MANIFEST.MF` entry in the JAR file, you can use the Apache Felix Maven Bundle plugin, which is added to the `pom.xml` file under the `<build>` section:

```

<build>
  <plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
      <instructions>
        <Bundle-Name>${project.artifactId}</Bundle-Name>
        <Bundle-SymbolicName>riderautoparts-osgi</Bundle-
SymbolicName>
        <Export-Package>
          camelinaction,
          camelinaction.inventory
        </Export-Package>
        <Import-Package>*</Import-Package>
        <Implementation-Title>Rider Auto Parts
OSGi</Implementation-Title>
        <Implementation-Version>${project.version}</Implementation-Version>
      </instructions>
    </configuration>
  </plugin>
</build>

```

The interesting part of `maven-bundle-plugin` is its ability to set the packages to be imported and exported. The plugin is set to export two packages: `camelinaction` and `camelinaction.inventory`. The `camelinaction` package contains the `InventoryRoute` Camel route, and it needs to be accessible by Camel so it can load the routes when the application is started. The `camelinaction.inventory` package contains the generated source files needed by Apache CXF when it exposes the web service.

In terms of imports, the preceding code uses an asterisk, which means the bundle plugin will figure it out by scanning the source for packages used. When needed, you can specify the imports by package name.

The book's source code contains this example in the `chapter15/osgi` directory. If you run the `mvn package` goal, you can see the `MANIFEST.MF` entry being generated in the `target/classes/META-INF` directory.

You've now set up Maven to build the JAR file as an OSGi bundle, which can be deployed to the container. The next step is to download and install Apache Karaf.

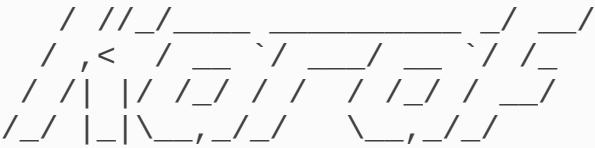
15.5.2 INSTALLING AND RUNNING APACHE KARAF

For this example, you can download and install the latest version of Apache Karaf from <http://karaf.apache.org>. (At the time of writing, this was Apache Karaf 4.1.2.) Installing is just a matter of extracting the zip or tar.gz file.

To run Apache Karaf, start it from the command line using one of these two commands:

```
bin/karaf      (Linux/Unix)  
bin\karaf.bat (Windows)
```

That should start up Karaf and display a logo when it's ready, like this:

```
[janstey@ghost apache-karaf-4.1.2]$ ./bin/karaf  
  
Apache Karaf (4.1.2)  
  
Hit '<tab>' for a list of available commands  
and '[cmd] --help' for help on a specific command.  
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to  
shutdown Karaf.  
  
karaf@root(>)
```

Running Karaf like this puts it in a shell mode, where you can enter commands to manage the container.

Now you need to install Camel and Apache CXF before you install the Rider Auto Parts application. Karaf makes installing easier using *features*, which are like *super bundles* that contain a set of bundles installed as a group. Installing features requires

you to add the Camel feature descriptions to Karaf, which you do by typing the following command in the shell:

```
feature:repo-add camel 2.20.1
```

You can then type `feature:list` to see all the features available to be installed. Among these should be several Camel-related features. To narrow the list, you can use `grep` just as you would on the command line:

```
feature:list | grep camel
```

The example application requires the `http`, `camel`, and `camel-cxf` features. Type these commands in the shell:

```
feature:install http  
feature:install camel  
feature:install camel-cxf
```

TIP You can type `bundle:list` to see which bundles have already been installed and their status. The shell has autocompletion, so you can press Tab to see the possible choices. For example, type `bundle` and then press Tab to see the choices.

Let's look at how the Rider Auto Parts application is modified for OSGi deployment.

15.5.3 USING AN OSGI BLUEPRINT-BASED CAMEL ROUTE

The most popular way to deploy Camel routes to an OSGi container such as Apache Karaf is in a Blueprint XML file. *Blueprint* is a dependency-injection framework for OSGi and part of the OSGi specification. In Karaf, the blueprint implementation comes from the Apache Aries project.

Camel routes defined in Blueprint XML look pretty much identical to their equivalents in Spring XML. That's why in many parts of this book we refer to this as the *XML DSL*. But let's call

out some differences in our Rider Auto Parts application so you get the full picture. First, the XML file is stored under the OSGI-INF/blueprint directory, which is different from Spring's META-INF/spring. Within the file, the XML looks close to the Spring equivalent:

```
<blueprint
    xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

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

http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
            http://camel.apache.org/schema/blueprint/cxf

http://camel.apache.org/schema/blueprint/cxf/camel-cxf.xsd
            http://camel.apache.org/schema/blueprint

http://camel.apache.org/schema/blueprint/camel-
blueprint.xsd">

    <!-- CXF endpoint we expose -->
    <cxf:cxfEndpoint id="inventoryEndpoint"
        address="http://localhost:9000/inventory"
        serviceClass="camelinaction.inventory.InventoryEndpoint"/>

    <!-- the order bean contain all the business logic -->
    <bean id="inventoryService"
        class="camelinaction.InventoryService"/>

    <!-- the route builder -->
    <bean id="inventoryRoute"
        class="camelinaction.InventoryRoute"/>

    <!-- Camel -->
    <camelContext id="myCamelContext"
        xmlns="http://camel.apache.org/schema/blueprint">
        <routeBuilder ref="inventoryRoute"/>
    </camelContext>

</blueprint>
```

All differences are highlighted in bold. As you can see, they're mainly just changes to XML namespaces and schemas.

You're now ready to deploy the Rider Auto Parts application.

15.5.4 DEPLOYING THE EXAMPLE

Karaf can install OSGi bundles from various sources, such as the filesystem or the local Maven repository.

To install using Maven, you first need to install the application in the local Maven repository, which you can easily do by running `mvn install` from the chapter15/osgi directory. After the JAR file has been copied to the local Maven repository, you can deploy it to Apache Karaf using the following command from the shell:

```
bundle:install mvn:com.camelinaction/riderautoparts-osgi/2.0.0
```

Upon installing any JAR to Karaf (a JAR file is known as a *bundle* in OSGi terms), Karaf will output on the console the bundle ID it has assigned to the installed bundle, as shown here:

```
Bundle ID: 133
```

You can then type `bundle:list` to see the application being installed:

```
karaf@root()> bundle:list
START LEVEL 100 , List Threshold: 50
 ID | State      | Lvl | Version | Name
-----
 57 | Active     | 50  | 2.20.1  | camel-blueprint
 58 | Active     | 50  | 2.20.1  | camel-catalog
 59 | Active     | 50  | 2.20.1  | camel-commands-core
 60 | Active     | 50  | 2.20.1  | camel-core
 61 | Active     | 50  | 2.20.1  | camel-cxf
 62 | Active     | 50  | 2.20.1  | camel-cxf-transport
 63 | Active     | 50  | 2.20.1  | camel-spring
 64 | Active     | 80  | 2.20.1  | camel-karaf-commands
133 | Installed | 80  | 2.0.0   | riderautoparts-osgi
```

Notice that the application isn't started. You can start it by

entering `bundle:start 133`, which changes the application's status when you do a `bundle:list` again:

```
133 | Active | 80 | 2.0.0 | riderautoparts osgi
```

The application is now running in the OSGi container.

TIP You can install and start the bundle in a single command using the `-s` option on the `bundle:install` command, like this:
`bundle:install -s mvn:com.camelinaction/riderautoparts-osgi/2.0.0.`

How can you test that it works? Start by checking the log with the `log:display` command. Among other things, it should indicate that Apache Camel has been started:

```
2016-12-20 20:02:07,174 | INFO | nsole user karaf | BlueprintCamelContext | 60 - org.apache.camel.camel-core - 2.20.1 | Apache Camel 2.20.1 (CamelContext: myCamelContext) started in 0.496 seconds
```

You can then use SoapUI to send a test request. The WSDL file is available at <http://localhost:9000/inventory?wsdl>.

When you're done testing the application, you may want to stop the OSGi container, which you can do by executing the `shutdown` command from the shell.

TIP You can tail the Apache Karaf log file using `tail-f log/karaf.log`. Note that this isn't done from within the Karaf shell but from the regular shell on your operating system. If you prefer to stay within Karaf, the Karaf shell also provides a `log:tail` command.

Taking our example a little further, let's look at using an OSGi-managed service factory to spin up multiple route instances

solely by modifying configuration.

15.5.5 USING A MANAGED SERVICE FACTORY TO SPIN UP ROUTE INSTANCES

Managed service factories (MSFs) in OSGi are a useful abstraction in dealing with multiple services that vary by only a few parameters. Going back to our Rider Auto Parts example, let's say you need to spin up several instances of the file inventory route, with each instance consuming from a different directory. Because only the file URI would be changing, it wouldn't make sense to create a new RouteBuilder class for each variation. A much more efficient way to do this in OSGi is to have a template route that accepts a few parameters and then have an MSF spin up a new instance based on configuration updates. For example, if you extract the file consumer route into its own template class, you'd have a reusable route:

```
public class FileInventoryRoute extends RouteBuilder {  
  
    private String inputPath;  
    private String routeId;  
  
    @Override  
    public void configure() throws Exception {  
        from(inputPath)  
            .routeId(getRouteId())  
            .split(body().tokenize("\n"))  
                .convertBodyTo(UpdateInventoryInput.class)  
                .to("direct:update");  
    }  
  
    // getter/setter omitted
```

Before this route would be useful, you'd have to set the `inputPath` and `routeId` fields. Looking at our Blueprint XML file from the previous section, you have to add only one extra element:

```
<bean id="camelServiceFactory"  
      class="camelaction.FileInventoryServiceFactory" init-  
      method="init">  
    <property name="bundleContext"  
      ref="blueprintBundleContext"/>
```

```
<property name="camelContext" ref="myCamelContext"/>
</bean>
```

This starts up your MSF `FileInventoryServiceFactory`, which can spin up additional instances of your `FileInventoryRoute` template route. The service factory is shown in the following listing.

Listing 15.9 A managed service factory for spinning up `FileInventoryRoute` instances

```
public class FileInventoryServiceFactory
    implements ManagedServiceFactory {
```

A service factory must implement
`org.osgi.service.cm.ManagedServiceFactory`

```
    private static final Logger LOG =
LoggerFactory.getLogger(FileInventoryServiceFactory.class);

    private CamelContext camelContext;
    private BundleContext bundleContext;
    private Map<String, FileInventoryRoute> routes
        = new HashMap<String, FileInventoryRoute>();
```

A map between persistent identifiers (PIDs) and routes

```
    private ServiceRegistration registration;

    @Override
    public String getName() {
        return "FileInventoryRouteCamelServiceFactory";
    }

    @SuppressWarnings("unchecked")
    public void init() {
        Dictionary properties = new Properties();
        properties.put( Constants.SERVICE_PID,
```

Register the service factory and provide the factory PID to watch for configuration

```
        "camelaction.fileinventoryroutefactory");
registration = bundleContext.registerService(
    ManagedServiceFactory.class.getName(),
    this, properties);

LOG.info("FileInventoryRouteCamelServiceFactory
ready to accept " +
    "new config with
PID=camelaction.fileinventoryroutefactory-xxx");
}

@Override
public void updated(String pid,
```

The updated method is called when configuration changes

```
Dictionary<String, ?> properties)
throws ConfigurationException {

String path = (String) properties.get("path");
```

The sole configuration property you'll be using for route instances

```
LOG.info("Updating route for PID=" + pid + " with
new path=" + path);

deleted(pid);
```

Need to remove the old route before adding the updated one

```
// now we create a new route with update path
FileInventoryRoute newRoute = new
FileInventoryRoute();
newRoute.setInputPath(path);
newRoute.setRouteId("file-" + pid);

try {
    camelContext.addRoutes(newRoute);
```

Add the new route to the CamelContext

```

        } catch (Exception e) {
            LOG.error("Failed to add route", e);
        }
        routes.put(pid, newRoute);
    }

@Override
public void deleted(String pid) {

```

The deleted method is called when the corresponding configuration is deleted

```

    LOG.info("Deleting route with PID=" + pid);

    try {
        FileInventoryRoute route = routes.get(pid);
        if (route != null) {
            camelContext.stopRoute(route.getRouteId());

            camelContext.removeRoute(route.getRouteId());
            routes.remove(pid);
        }
    } catch (Exception e) {
        LOG.error("Failed to remove route", e);
    }
}

...
}
```

To explain what's going on here, you need to know about how configuration works in an OSGi container such as Karaf. Configuration is managed by the Configuration Admin service. Each group of configuration items tracked by Configuration Admin has a unique persistent identifier (PID). For our MSF, you assign a factory PID

camelaction.fileinventoryroutefactory, which means Configuration Admin will watch for configuration with PIDs such as camelaction.fileinventoryroutefactory-foo, camelaction.fileinventoryroutefactory-bar, and so forth. They just have to start with the factory PID for your MSF to get picked up. To put it more concretely, in the Karaf distribution, if you add a camelaction.fileinventoryroutefactory-path1.cfg file

to the etc directory with these contents

```
path=file://target/inventory/updates
```

that would get picked up by Configuration Admin, which would in turn call the updated method on our MSF. This would add a new route to CamelContext. In the logs, you'd see something like this:

```
2016-12-21 19:31:05,127 | INFO  | f7-466f8727ed59) |  
BlueprintCamelContext           | 60 -  
org.apache.camel.camel-core - 2.20.1 | Route: file-  
camelinaction.fileinventoryroutefactory.32f4ffa3-e2cc-4930-  
a9f7-466f8727ed59 started and consuming from:  
file://target/inventory/updates
```

You could add as many extra configuration files as you want—one for each distinct route. This also doesn't all have to be done through configuration files; Configuration Admin can be controlled via Karaf config:* commands, JMX, or even programmatically.

Check out the MSF example in the chapter15/osgi-msf directory of the book's source code.

You've now seen how to deploy a Camel application into an OSGi container. [Table 15.6](#) lists the pros and cons of deploying Camel in an OSGi container.

Table 15.6 Pros and cons of using OSGi as a deployment strategy

Pros	Cons
<ul style="list-style-type: none">• Uses OSGi for modularity• Provides classloader isolation and hot deployment	<ul style="list-style-type: none">• Involves a learning curve for OSGi• Unsupported third-party frameworks; some frameworks have yet to become OSGi compliant• Requires extra effort to decide what package imports and exports to use for your module

We've only scratched the surface of OSGi in this chapter. If you go down that path, you'll need to pick up other books, because OSGi is a big concept to grasp and master. It's also powerful and allows you to create dynamic applications. On the other hand, the path of the web application is the beaten track, and plenty of materials and people can help you if you come up against any problems.

15.6 Camel and CDI

Contexts and Dependency Injection (CDI) is the part of the Java EE platform focused on the following:

- Dependency injection
- Lifecycle management of stateful objects bound to contexts

It's easy to get started with CDI. You typically need to add a single dependency to your project:

```
<dependency>
    <groupId>javax.enterprise</groupId>
    <artifactId>cdi-api</artifactId>
    <version>1.2</version>
    <scope>provided</scope>
</dependency>
```

Use CDI annotations and then deploy to a container that supports CDI such as WildFly, WebSphere, or Apache TomEE. Many containers support CDI using the reference implementation, the JBoss Weld project.

Instead of making you manually instantiate and wire CamelContexts and routes together, Camel (the camel-cdi component in particular) automatically deploys and configures CamelContext for you. It searches for any routes as well and adds them to the context. Common Camel services such as endpoints, ProducerTemplates, TypeConverters, and even CamelContext itself are injectable via CDI annotations.

We first covered CDI in chapter 7, so many of the basic

concepts are discussed there. Testing CDI applications was discussed in chapter 9. If you’re unfamiliar with CDI, you should review those chapters first. This section focuses on deployment to the WildFly container.

NOTE It’s possible to deploy CDI-based routes in OSGi, but this support is deprecated in Camel as of version 2.19.0, so we don’t cover it here. You may still be able to find an example in the distribution in the examples/camel-example-cdi-osgi directory.

Back in section 15.4.3 we discussed how to deploy Camel routes to the WildFly container. We also mentioned how to use an extension project called WildFly-Camel to make deployment easier. We’ll be assuming a WildFly-Camel-enhanced container as well in this section.

First, let’s go over what you have to add in your Maven pom.xml file. In the dependencies section, you have this:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-cdi</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.enterprise</groupId>
    <artifactId>cdi-api</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.wildfly.camel</groupId>
    <artifactId>wildfly-camel-subsystem-core</artifactId>
    <scope>provided</scope>
</dependency>
```

The first two are probably obvious if you’ve read the previous sections that covered CDI. The last one is for an annotation specifically for WildFly-Camel. Although not absolutely necessary, it’s helpful to import the WildFly-Camel BOM so that

you don't need to specify any dependency versions. You can do so as follows:

```
<dependencyManagement>
  <dependencies>
    <!-- WildFly Camel -->
    <dependency>
      <groupId>org.wildfly.camel</groupId>
      <artifactId>wildfly-camel</artifactId>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.wildfly.camel</groupId>
      <artifactId>wildfly-camel-patch</artifactId>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Moving on to our route, you don't have to add any CDI annotations to it for camel-cdi to pick it up. For WildFly-Camel to recognize it, though (as of version 5.0.0), you need either an `org.wildfly.extension.camel.CamelAware` or `org.apache.camel.cdi.ContextName` annotation. The route is shown here:

```
import org.apache.camel.builder.RouteBuilder;
import org.wildfly.extension.camel.CamelAware;
import camelaction.inventory.UpdateInventoryInput;

@CamelAware
public class InventoryRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        // this is the file route which is started 2nd last
        from("file://target/inventory/updates")
            .routeId("file").startupOrder(2)
            .split(body().tokenize("\n"))
                .convertBodyTo(UpdateInventoryInput.class)
                .to("direct:update");

        // this is the shared route which then must be
        started first
        from("direct:update")
```

```

        .routeId("update").startupOrder(1)
        .to("bean:inventoryService?
method=updateInventory");
    }
}

```

As you can see, the route doesn't look that different from before except for the `@CamelAware` annotation. In previous examples, the inventory service was defined in Spring or Blueprint as follows:

```
<bean id="inventoryService"
class="camelaction.InventoryService"/>
```

But here, you have no such definition. With CDI, you can create named beans via annotations. Let's look at your `InventoryService` bean in action:

```

@Named("inventoryService")
public class InventoryService {

    private Random ran = new Random();

    public String xmlToCsv(UpdateInventoryInput input) {
        return input.getSupplierId() + "," +
input.getPartId()
            + "," + input.getName() + "," +
input.getAmount();
    }

    public UpdateInventoryOutput replyOk() {
        UpdateInventoryOutput ok = new
UpdateInventoryOutput();
        ok.setCode("OK");
        return ok;
    }

    public void updateInventory(UpdateInventoryInput input)
throws Exception {
        int sleep = ran.nextInt(1000);
        Thread.sleep(sleep);

            System.out.println("Inventory " + input.getPartId()
+ " updated");
    }

}

```

Here you can see that you use the `javax.inject.Named` annotation to create a named bean instance you can refer to in your Camel route. The `camel-cdi` component starts up `org.apache.camel.cdi.CdiCamelRegistry` to hold such beans in `CamelContext`. To try this example for yourself, change to the `chapter15/cdi` directory of the book's source and run this command:

```
mvn clean install -Pdeploy
```

This deploys the example to a locally running instance of WildFly (if one is running). One handy tip for deploying during development is to use `wildfly-maven-plugin` as we have in this example. To use this plugin, you can add something like the following to your Maven `pom.xml` file:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.wildfly.plugins</groupId>
      <artifactId>wildfly-maven-plugin</artifactId>
      <configuration>
        <skip>${deploy.skip}</skip>
      </configuration>
      <executions>
        <execution>
          <id>wildfly-deploy</id>
          <phase>install</phase>
          <goals>
            <goal>deploy-only</goal>
          </goals>
        </execution>
        <execution>
          <id>wildfly-undeploy</id>
          <phase>clean</phase>
          <goals>
            <goal>undeploy</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

<!-- Profiles -->
```

```
<profiles>
  <profile>
    <id>deploy</id>
    <properties>
      <deploy.skip>false</deploy.skip>
    </properties>
  </profile>
</profiles>
```

Notice that it's turned off by default (because starting WildFly is a separate manual step), so you have to provide the `deploy` profile to enable it.

15.7 Summary and best practices

This chapter explored the internal details of the way Camel starts up. You learned which options you can control and whether routes should be autostarted. You also learned how to dictate the order in which routes should be started.

More importantly, you learned how to shut down a running application in a reliable way without compromising the business. You learned about Camel's graceful shutdown procedures and what you can do to reorder your routes to ensure a better shutdown process, when needed. You also learned how to stop and shut down routes at runtime. You can do this programmatically, which allows you to fully control when routes are operating and when they aren't.

In the second part of this chapter, you explored the art of deploying Camel applications as standalone Java applications, as web applications, in Java EE containers, as CDI applications, and by running Camel in an OSGi container. Remember that the deployment strategies covered in this book aren't all of your options. For example, we covered several options popular with microservices in chapter 7 and will cover Docker containers in chapter 18.

Here are some pointers to help you with running and deployment:

- *Ensure reliable shutdown*—Take the time to configure and test that your application can be shut down in a reliable manner. Your application is bound to be shut down at some point, whether for planned maintenance, upgrades, or unforeseen problems. In those situations, you want the application to shut down in a controlled manner without negatively affecting your business.
- *Use an existing runtime environment*—Camel is agile, flexible, and can be embedded in whatever production setup you may want to use. Don't introduce a new production environment just for the sake of using Camel. Use what's already working for you, and test early in the project that your application can be deployed and run in the environment.

In the next chapter, you'll tour Camel's extensive monitoring and management facilities.

16

Management and monitoring

This chapter covers

- Monitoring Camel instances
- Tracking application activities
- Using notifications
- Managing Camel applications with JMX and REST
- Understanding and using the Camel management API
- Gathering runtime performance statistics
- Using Dropwizard metrics with Camel
- Developing custom components for management

Applications in production are often critical for businesses. That's especially true for applications that sit at an intermediate tier and integrate all the business applications and partners. Camel is often in this role.

To help ensure high availability, your organization must monitor its production applications. By doing so, you can gain important insight into the applications and foresee trends that otherwise could cause business processes to suffer. In addition, monitoring helps with related issues such as operations reporting, service-level agreement (SLA) enforcement, and audit trails.

It's also vital for the operations staff to be able to fully manage the applications. For example, if an incident occurs, staff may need to stop parts of the application from running while the incident investigations occur. You'll also need management capabilities to carry out scheduled maintenance or upgrades of your applications.

Management and monitoring are often two sides of the same coin. For example, management tooling includes monitoring capabilities in a single coherent dashboard, allowing a full overview for the operations staff.

This chapter reviews various strategies for monitoring your Camel applications. We'll first cover the most common approach, which is to check on the health of those applications. Then we'll look at the options for tracking activity and managing those Camel applications.

16.1 Monitoring Camel

It's standard practice to monitor systems with periodic health checks. For people, checking one's health involves measuring parameters at regular intervals, such as pulse, temperature, and blood pressure. By checking over a period of time, you know not only the current values but can also spot trends, such as whether your temperature is rising. All together, these data give insight into the health of the person.

For a software system, you can gather system-level data such as CPU load, memory usage, and disk usage. You can also collect application-level data, such as message load, response time, and many other parameters. This data tells you about the health of the system. Checks on the health of Camel applications can occur at three levels:

- *Network level*—This is the most basic level, where you check that the network connectivity is working.
- *JVM level*—At this level, you check the JVM that hosts the Camel application. The JVM exposes a standard set of data using the

JMX technology.

- *Application level*—Here you check the Camel application using JMX or other techniques.

To perform these checks, you need various tools and technologies. The Simple Network Management Protocol (SNMP) enables both JVM and system-level checks. Java Management Extensions (JMX) is another technology that offers similar capabilities to SNMP. You might use a mix of both: SNMP is older, more mature, and often used in large system-management tools including established technologies such as IBM Tivoli and HP OpenView. JMX, on the other hand, is a pure Java standard supported by open source tools such as Jolokia, Hawkular, Prometheus, and Nagios. The following sections go over the three levels and offer approaches you can use for performing automatic and periodic health checks on your Camel applications.

16.1.1 CHECKING HEALTH AT THE NETWORK LEVEL

The most basic health check you can do is to check whether a system is alive. You may be familiar with the ping command, which you use to send a ping request to a remote host. Camel doesn't provide a ping service out of the box, but creating such a service is easy. The ping service reveals only whether Camel is running, but that will do for a basic check.

Suppose you've been asked to create such a ping service for Rider Auto Parts. The service is to be integrated with the existing management tools. You choose to expose the ping service over HTTP, which is a universal protocol that the management tool can easily use, a scenario illustrated in [figure 16.1](#).

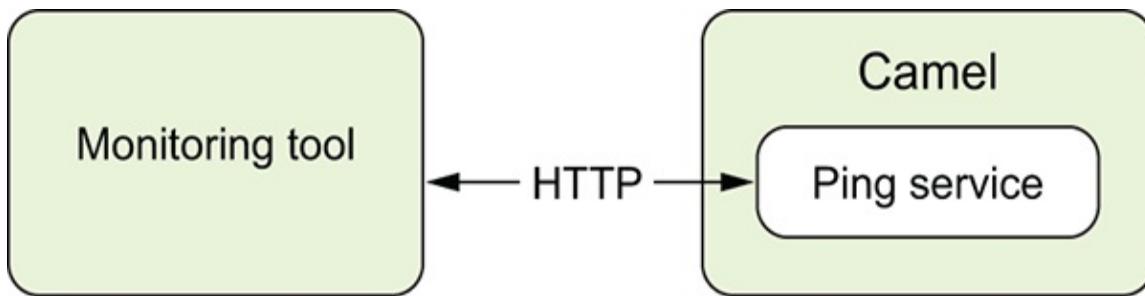


Figure 16.1 A monitoring tool monitors Camel with a ping service by sending periodic HTTP GET requests.

Implementing the service in Camel is easy when using the Jetty component. All you have to do is expose a route that returns the response, as follows:

```
from("jetty:http://0.0.0.0:8080/ping").transform(constant("PONG\n"));
```

When the service is running, you can invoke an HTTP GET, which should return the PONG response.

You can try this on your own with the book's source code. In the chapter16/health directory, invoke this Maven goal:

```
mvn compile exec:java
```

Then invoke the HTTP GET using either a web browser or the curl command:

```
$ curl http://0.0.0.0:8080/ping
PONG
```

The ping service can be enhanced to use the JVM and Camel APIs to gather additional data about the state of the internals of your application.

USING JOLOKIA AS PING SERVICE

Instead of building your own Camel route as a ping service, you can use Jolokia. The Jolokia Java JVM agent, upon starting the Camel application, will bootstrap Jolokia automatically.

Jolokia

Jolokia is an excellent library that makes JMX management fun again. At its core, it's an HTTP/JSON bridge for remote JMX access. It provides a Java agent that makes it easy to integrate into a JVM without any code changes. You can find more details about Jolokia at its website:
<https://jolokia.org>.

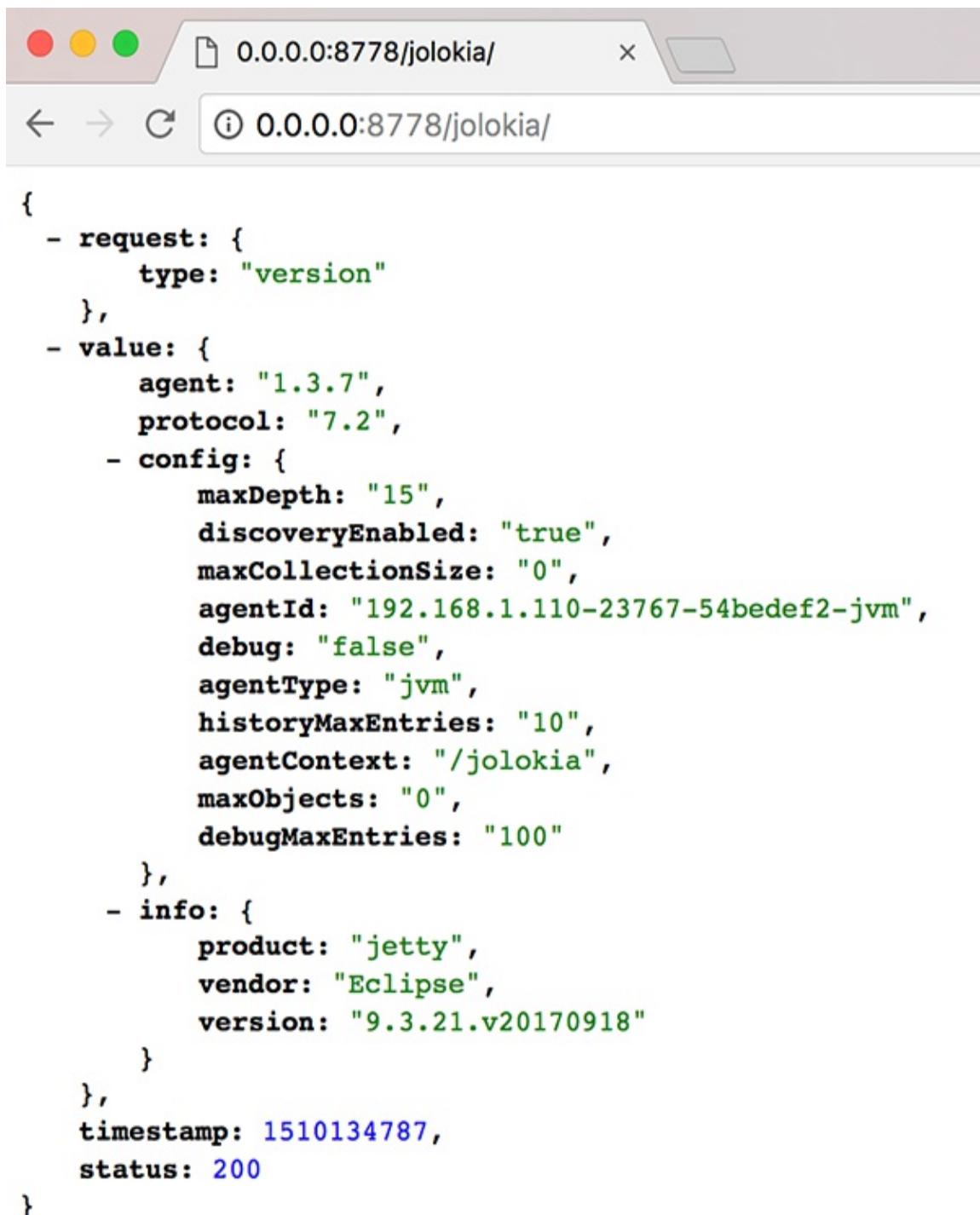
You can try this example on your own with the book's source code. In the chapter16/jolokia directory, invoke this Maven goal:

```
mvn compile exec:exec
```

Then invoke the HTTP GET using either a web browser or the curl command (make sure to include the trailing slash):

```
curl http://0.0.0.0:8778/jolokia/
```

Jolokia returns a JSON response with information about the Jolokia agent running in the JVM, as shown in [figure 16.2](#).



A screenshot of a web browser window displaying a JSON configuration object. The URL bar shows "0.0.0.0:8778/jolokia/". The JSON content includes fields like "request", "value", "info", and "timestamp".

```
{
  - request: {
      type: "version"
    },
  - value: {
      agent: "1.3.7",
      protocol: "7.2",
      - config: {
          maxDepth: "15",
          discoveryEnabled: "true",
          maxCollectionSize: "0",
          agentId: "192.168.1.110-23767-54bedef2-jvm",
          debug: "false",
          agentType: "jvm",
          historyMaxEntries: "10",
          agentContext: "/jolokia",
          maxObjects: "0",
          debugMaxEntries: "100"
        },
      - info: {
          product: "jetty",
          vendor: "Eclipse",
          version: "9.3.21.v20170918"
        }
    },
  timestamp: 1510134787,
  status: 200
}
```

Figure 16.2 Web browser showing the returned information from calling the Jolokia agent on the given URL

Jolokia allows you to access JMX attributes that you can use to read the Uptime attribute from the Camel application. The following URL retrieves this information:

```
http://localhost:8778/jolokia/read/org.apache.camel:context=camel-1  
,name=%22camel-1%22,type=context/Uptime
```

And Jolokia will respond with JSON, where you can see that the Uptime value is 8 minutes:

```
{"timestamp":1510141224,"status":200,"request":  
{"mbean":"org.apache.camel:context=camel-1,name=\"camel-  
1\"",  
"type=context","attribute":"Uptime","type":"read"}, "value":  
"8 minutes"}
```

TIP Instead of reading the Uptime attribute, you can try some of the many attributes that Camel exposes in JMX, such as CamelVersion and ExchangesTotal.

We'll now leave Jolokia for a bit and continue. Another use for the ping service is when using a load balancer in front of multiple instances of Camel applications. This is often done to address high availability, as shown in [figure 16.3](#). The load balancer will call the ping service to assess whether the particular Camel instance is ready for regular service calls.

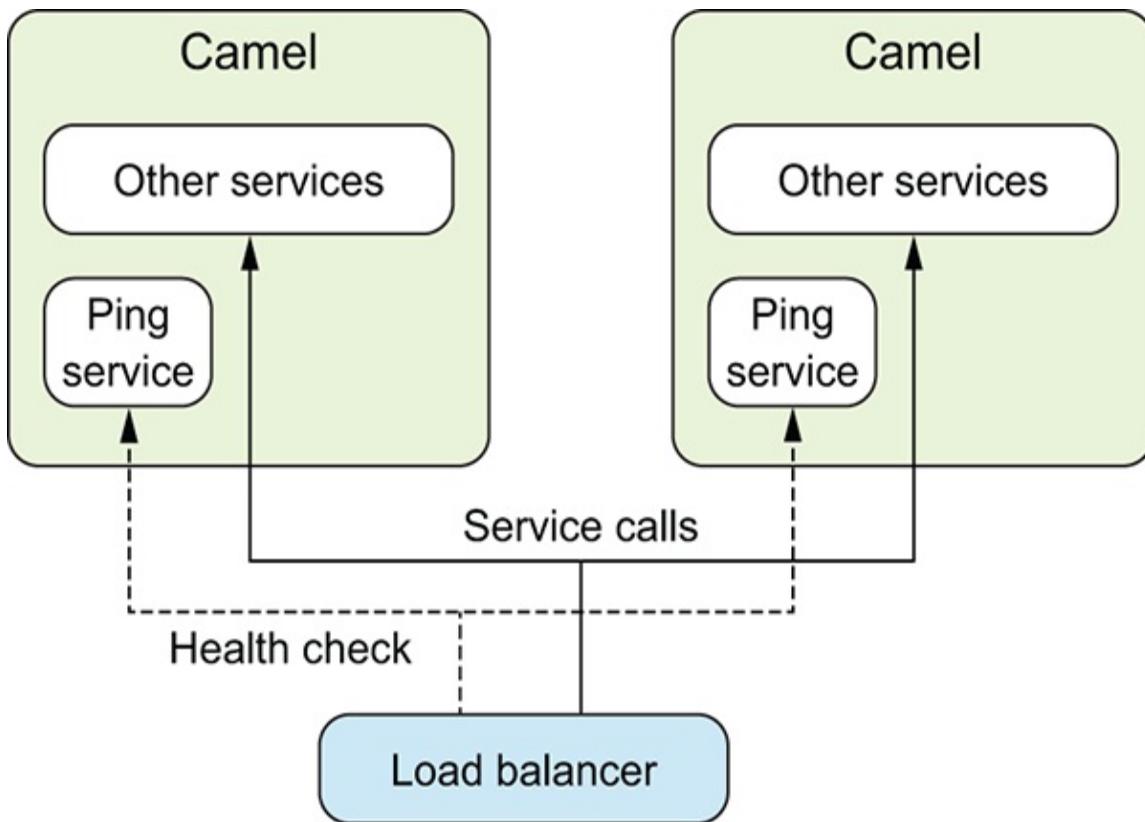


Figure 16.3 The load balancer uses health checks to ensure connectivity before it lets the service calls pass through.

Network-level checks offer a quick and coarse assessment of the system’s state of health. Let’s move on to the JVM level, where you monitor Camel using JMX.

16.1.2 CHECKING HEALTH LEVEL AT THE JVM LEVEL

SNMP is a standard for monitoring network-attached devices. It’s traditionally used to monitor the health of servers at the OS level by checking parameters such as CPU load, disk space, memory usage, and network traffic, but it can also be used to check parameters at the application level, such as the JVM.

Java has a built-in SNMP agent that exposes general information, such as memory and thread usage, and issues notifications on low-memory conditions. This allows you to use existing SNMP-aware tooling to monitor the JVM where Camel is running. A wide range of commercial and open source

monitoring tools also use SNMP. Some are simple and have a shell interface, and others have a powerful GUI. You may work in an organization that already uses a few monitoring tools, so make sure these tools can be used to monitor your Camel applications as well.

The SNMP agent in the JVM is limited to exposing data at only the JVM level; it can't be used to gather information about the Java applications that are running. JMX, in contrast, is capable of monitoring and managing both the JVM and the applications running on it.

In the next section, we'll look at how to use JMX to monitor Camel at the JVM and application levels.

16.1.3 CHECKING HEALTH AT THE APPLICATION LEVEL

Camel provides JMX monitoring and management out of the box in the form of an agent that uses the JMX technology. This is illustrated in [figure 16.4](#).

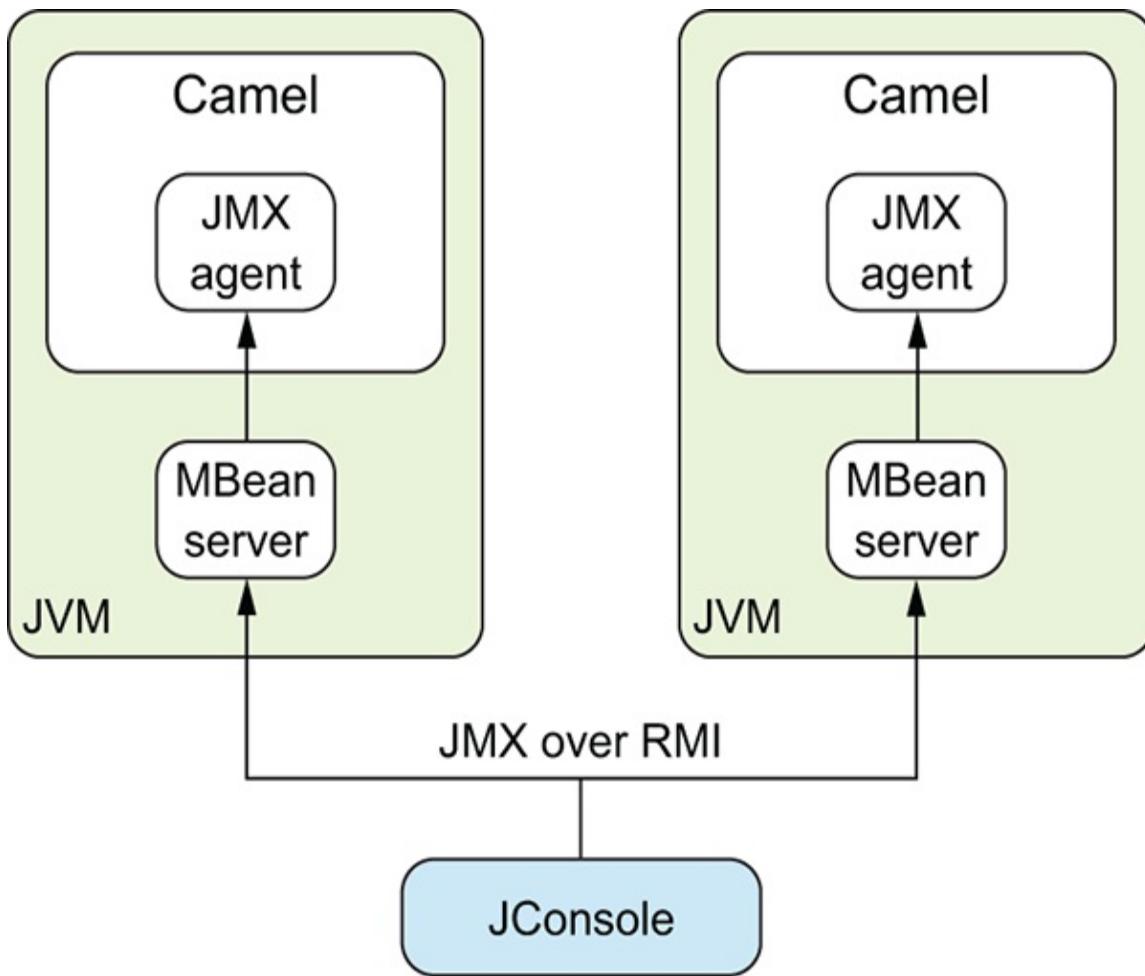


Figure 16.4 JConsole connects remotely to an MBean server inside the JVM, which opens up a world of in-depth information and management possibilities for Camel instances.

The JMX agent remotely exposes (over Remote Method Invocation) a wealth of standard details about the JVM as well as some Camel information. The former comes standard from the JDK, and the latter is provided by Camel. The most prominent feature the Camel JMX agent offers is the ability to remotely control the lifecycle of any service in Camel. For example, you can stop routes and later bring those routes into action again. You can even shut down Camel itself.

How do you use JMX with Camel? Camel comes preconfigured with JMX enabled at the developer level, by which we mean that Camel allows you to connect to the JVM from the same localhost where the JVM is running. If you need to manage Camel from a

remote host, you need to explicitly enable this in Camel.

We think this is important to cover thoroughly, so we devote the next section to this topic.

16.2 Using JMX with Camel

Camel comes with JMX enabled out of the box. When Camel starts, it logs at the `INFO` level whether JMX is enabled or not:

```
2017-08-06 15:00:51,361 [viceMain.main()] INFO  
ManagedManagementStrategy - JMX is enabled
```

And when JMX is disabled:

```
2017-08-06 15:02:53,885 [viceMain.main()] INFO  
ManagedManagementStrategy - JMX is disabled
```

In this section, we'll look at two management tools that can manage Java applications using JMX. The first is JConsole, which is shipped out of the box in Java. The other tool is Jolokia, with hawtio used as the web console.

16.2.1 USING JCONSOLE TO MANAGE CAMEL

Java provides a JMX tool named JConsole. You'll use it to connect to a Camel instance and see what information is available.

TIP Java provides another management tool named JVisualVM that you can use as well. But you'd need to install the VisualVM-MBeans plugin to allow this tool to work with JMX Management Beans.

First, you need to start a Camel instance. You can do this from the chapter16/health directory using this Maven command:

```
mvn compile exec:java
```

Then, from another shell, you can start JConsole by invoking

jconsole.

When JConsole starts, it displays a window with two radio buttons. The Local radio button is used to connect to existing JVMs running on the same host. The Remote radio button is used for remote management, which we'll cover shortly. The Local option should already list the local running processes. You can find the correct process by hovering the mouse to display a tooltip with the full name. The process containing the word Launcher corresponds to the running application that was launched from Maven. Select this process, and you can click the Connect button to connect JConsole to the Camel instance.

Figure 16.5 shows the Camel management beans (MBeans) that are visible from JConsole.

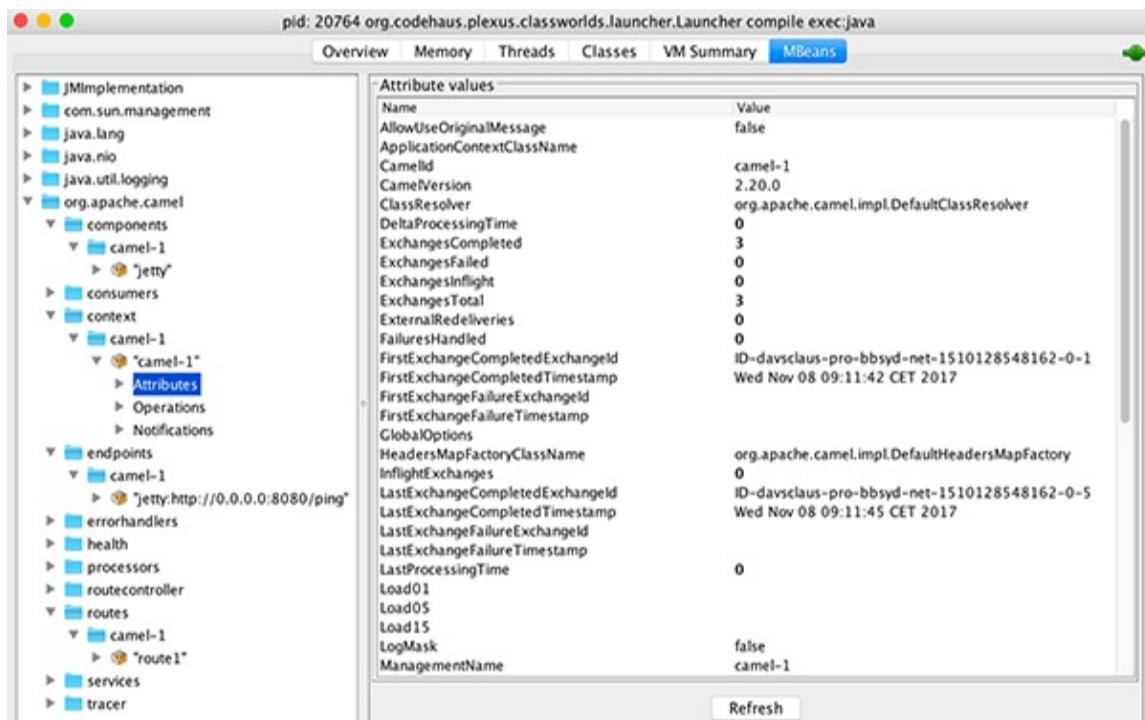


Figure 16.5 Camel registers numerous MBeans that expose internal details, such as usage statistics and management operations.

Camel registers many MBeans that expose statistics and operations for management. Those MBeans are divided into 12 categories, which are listed in table 16.1. Most MBeans expose a set of standard information and operations concerning things

such as lifecycle. We encourage you to spend a moment browsing the MBeans in JConsole to see what information they provide.

Table 16.1 Categories of exposed Camel MBeans

Category	Description
Components	Lists the components in use.
Consumers	Lists all the consumers for the Camel routes. Some consumers have additional information and operations, such as the JMS, SEDA, Timer, and File/FTP consumers.
Context	Identifies the <code>CamelContext</code> itself. This is the MBean you need if you want to shut down Camel via JMX.
Data formats	Lists the data formats in use.
Endpoints	Lists the endpoints in use.

nt s	
E rr or h a n d l e r s	Lists the error handlers in use. You can manage error handling at runtime, such as by changing the number of redelivery attempts or the delay between redeliveries.
E v e n t n o t i f i e r s	Lists the event notifiers in use. We'll cover the use of event notifiers in section 16.3.5.
P r o c e s s o r s	Lists all the processors (enterprise integration patterns) in use. The EIPs provide various runtime information and operations you can access. For example, the Content-Based Router EIP contains statistics indicating the number of times each predicate has matched.
P r o d u c e r s	Lists all the producers for the Camel routes. Some producers have additional information and operations such as JMS, SEDA, and File/FTP producers.
R o u t e s	Lists all the routes in use. Here you can obtain route statistics, such as the number of messages completed, failed, and so on.
S e r vi	Lists miscellaneous services in use.

c e s	
T hr e a d p o ol s	<p>Lists all the thread pools in use. Here you can obtain statistics about the number of threads currently active and the maximum number of threads that have been active. You can also adjust the core and maximum pool size of the thread pool.</p>
T ra c er	<p>Allows you to manage the Tracer service. The Tracer is a Camel-specific service that's used for tracing how messages are routed at runtime. We cover the use of the Tracer in detail in section 16.3.4.</p>

When you need to monitor and manage a Camel instance from a remote computer, you must enable remote management in Camel.

16.2.2 USING JCONSOLE TO REMOTELY MANAGE CAMEL

To be able to remotely manage Camel, you need to instruct Camel to register a JMX connector. That can be done in the following three ways:

- Using JVM properties
- Configuring ManagementAgent from Java
- Configuring the JMX agent from XML DSL

We'll go over each of these three methods in the following sections.

USING JVM PROPERTIES

By specifying the following JVM property on JVM startup, you can tell Camel to create a JMX connector for remote management:

```
-Dorg.apache.camel.jmx.createRmiConnector=true
```

If you do this, Camel will log, at `INFO` level on startup, the JMX service URL that's needed to connect. It looks something like this:

```
2017-08-06 15:58:37,264 [viceMain.main()] INFO  
ManagedManagementStrategy  
- JMX is enabled  
2017-08-06 15:58:37,267 [viceMain.main()] INFO  
DefaultManagementAgent  
- ManagementAgent detected JVM system properties:  
{org.apache.camel.jmx.  
createRmiConnector=true}  
2017-08-06 15:58:37,492 [99/jmxrmi/camel] INFO  
DefaultManagementAgent  
- JMX Connector thread started and listening at:  
service:jmx:rmi:///jndi/rmi://davsclaus-  
pro:1099/jmxrmi/camel
```

To connect to a remote JMX agent, you can use the Remote radio button from JConsole and enter the service URL listed in the log. By default, port 1099 is used, but this can be configured using the `org.apache.camel.jmx.rmiConnector.registryPort` JVM property.

CONFIGURING THE MANAGEMENTAGENT FROM JAVA

The `org.apache.camel.management.DefaultManagementAgent` class is provided by Camel as the default JMX agent. All you need to do is to configure the settings directly using the `ManagementAgent` interface, as highlighted here:

```
public class PingServiceMain {  
    public static void main(String[] args) throws Exception  
{  
        CamelContext context = new DefaultCamelContext();  
  
        context.getManagementStrategy().getManagementAgent()  
            .setCreateConnector(true);  
  
        context.addRoutes(new PingService());  
        context.start();
```

```
    }  
}
```

CONFIGURING A JMX AGENT FROM XML DSL

If you use XML DSL with Camel, configuring a JMX connector is even easier. All you have to do is add `<jmxAgent>` in the `<camelContext>`, as shown here:

```
<camelContext id="camel"  
xmlns="http://camel.apache.org/schema/spring">  
    <jmxAgent id="agent" createConnector="true"/>  
    ...  
</camelContext>
```

`<jmxAgent>` also offers a `registryPort` attribute that you can use to set a specific port number if the default port 1099 isn't suitable.

JMX: The good, bad, and ugly

JMX is a good technology for Java libraries and applications to expose a management API, and provide runtime metrics about the state of the JVM and application. But it has its serious drawbacks, as it's a Java-only technology that requires clients to use Java as well (or CORBA). Also, as you've witnessed, remote management requires using a connector port (usually 1099 as the default), and then the JVM assigns a random port as well that's in use. This is seriously not firewall friendly, as organizations have to open ports in their network for JMX remote management. For more details, see, for example, the Jolokia site: <https://jolokia.org/features/firewall.html>.

How can clients using another technology access and manage Java applications, which are also firewall friendly? A great answer is to use Jolokia.

16.2.3 USING JOLOKIA TO MANAGE CAMEL

Jolokia allows you to access JMX using HTTP. This ubiquitous technology can be accessed from a web browser, an HTTP client from the command line, or a web console such as hawtio.

TIP We recommend that you read about the features of Jolokia from its website: <https://jolokia.org/features-nb.html>.

Jolokia offers numerous agents providing Jolokia services in various environments. This chapter covers these three agents:

- WAR agent for deployment as a web application in an application server such as Apache Tomcat or WildFly
- OSGi agent for deployment in an OSGi container such as Apache Karaf or JBoss Fuse
- JVM agent as a generic agent using Java agent style

The WAR agent is the easiest to use, so let's start there.

USING THE JOLOKIA WAR AGENT

The WAR agent is used by deploying the agent in a web container such as Apache Tomcat or WildFly. You can either deploy the preexisting Jolokia WAR or embed Jolokia into your existing WAR application. In this section, we'll walk you through both approaches, starting by deploying the out-of-the box WAR in the following steps:

1. Start Apache Tomcat.
2. Download Jolokia WAR from <https://jolokia.org/download.html>.
3. Copy the downloaded WAR into the webapps directory of the running Apache Tomcat.
4. Check whether Jolokia is running by opening

<http://localhost:8080/jolokia-war-1.3.7> (assuming you're using version 1.3.7 of Jolokia).

5. If Jolokia is running, the web page should output a Jolokia status page (similar to [figure 16.2](#)).

By installing the Jolokia WAR once into Apache Tomcat, you can use Jolokia to manage any of the applications deployed in Tomcat. For example, if you deploy five Camel applications and three other applications, Jolokia can access all of those deployments.

Another approach is to embed Jolokia into your existing WAR application, which you may want to do to make a single deployment have *batteries included*.

EMBEDDING JOLOKIA WAR AGENT INTO EXISTING WAR APPLICATION

You can embed Jolokia into your existing WAR application by adding a dependency to Jolokia in the Maven pom.xml file:

```
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
    <version>1.3.7</version>
</dependency>
```

Then add Jolokia to the web.xml file to expose the Jolokia as an HTTP service as a servlet:

```
<servlet>
    <servlet-name>jolokia-agent</servlet-name>
    <servlet-class>org.jolokia.http.AgentServlet</servlet-
class>
</servlet>

<servlet-mapping>
    <servlet-name>jolokia-agent</servlet-name>
    <url-pattern>/jolokia/*</url-pattern>
</servlet-mapping>
```

The book's source code contains an example in the chapter16/jolokia-embedded-war that you can try using the following Maven goal:

```
mvn jetty:run
```

You can also build the example and deploy the WAR file to a web container such as Apache Tomcat using the following:

```
mvn clean install
```

Then copy the generated WAR file target/chapter16-jolokia-war.war to the webapps directory of Apache Tomcat.

If you're using Apache Tomcat with the default HTTP port 8080, you can access Jolokia using the following URL from a web browser, which should report back a status response from Jolokia:

```
http://localhost:8080/chapter16-jolokia-embedded-war/jolokia/
```

The principle of embedding a Jolokia WAR agent together with your Camel application in the same deployment unit is illustrated in [figure 16.6](#).

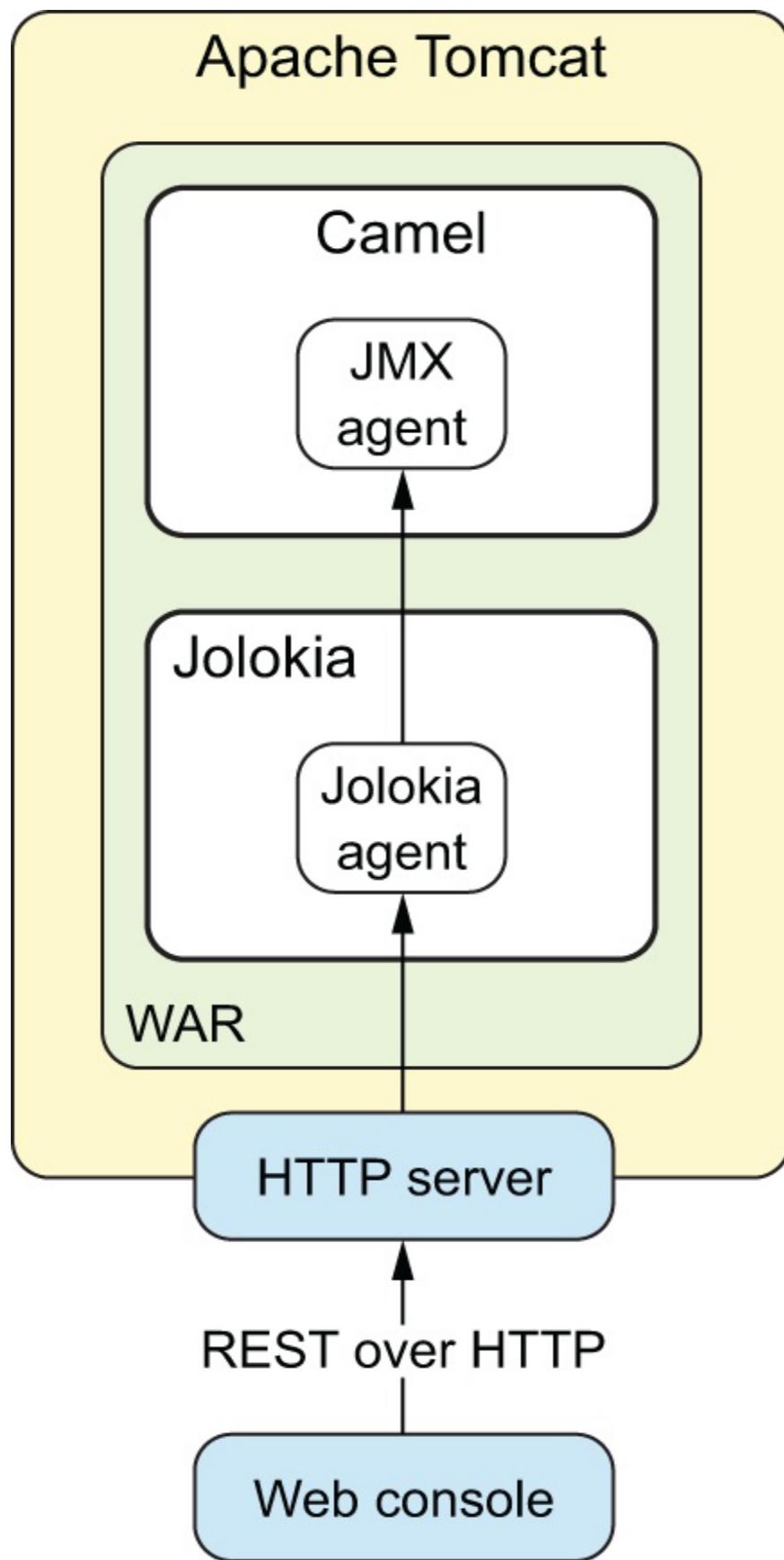


Figure 16.6 Jolokia is embedded together with Camel into the same WAR deployment unit, which is running inside Apache Tomcat. The Jolokia service is exposed using the HTTP server of Apache Tomcat, which the web console can access to obtain information about the running Camel application.

hawtio, the hot web console

As a cool web console, you can use hawtio. Try the example from chapter16/jolokia-embedded-war and deploy hawtio into Apache Tomcat. The hawtio console will autodetect the Camel applications running in the JVM, which allows you to use the plugin to visualize the running Camel routes as well as to manage them, and much more. Chapter 19 covers hawtio in more detail.

The next kind of agent is the OSGi agent, which works in OSGi containers such as Apache Karaf and JBoss Fuse. Because Jolokia is preinstalled in JBoss Fuse, we use Apache Karaf as a demonstration for installing and using Jolokia.

USING JOLOKIA WITH APACHE KARAF

Using Jolokia on Apache Karaf is easy because you can install Jolokia using the Karaf shell. After Karaf has been started, type the following in the shell (Karaf 4 onward):

```
feature:install war  
install -s mvn:org.jolokia/jolokia-osgi/1.3.7
```

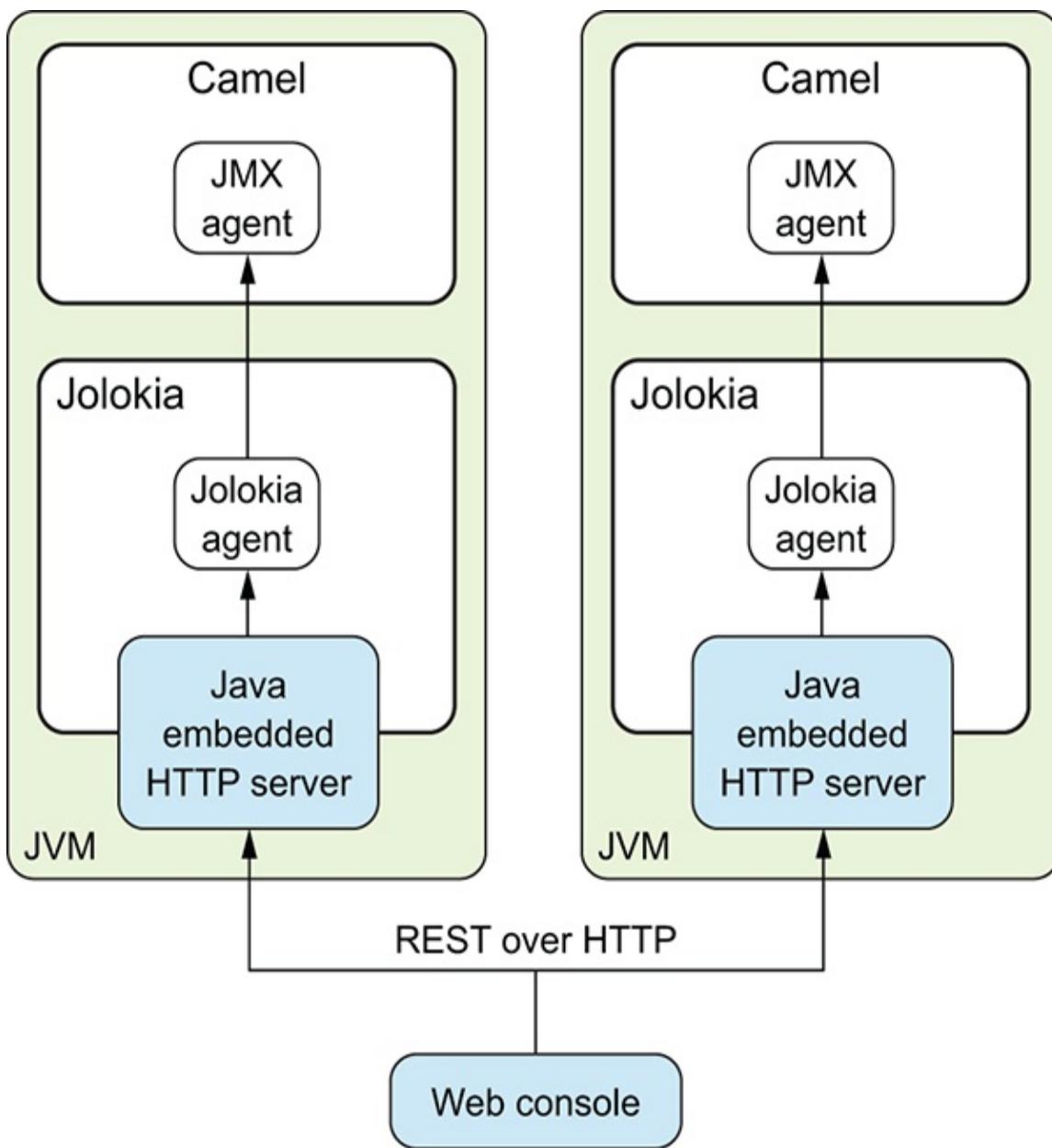
Jolokia is then available on the default Karaf HTTP port (8181) using the context path jolokia. You should be able to access Jolokia using the following URL:

```
http://localhost:8181/jolokia/
```

Okay, let's move on and talk about the last kind of Jolokia agent: the Java JVM agent.

USING JOLOKIA JAVA AGENT

Jolokia provides a JVM agent (a.k.a. *Java agent*) that doesn't need any special runtime environment because it uses the embedded HTTP server from Java itself (requires Java 1.6 onward). This agent is the most generic and can be used for any Java application. It's particularly useful when the other specialized agents don't fit. [Figure 16.7](#) illustrates how this agent works.



[Figure 16.7](#) A web console (or HTTP client) connects remotely to a JVM of

choice to manage using HTTP. The JVM embeds the HTTP server, and the Jolokia agent bridges HTTP to JMX so the web console can gather all JMX information from the Camel application.

The Jolokia Java agent is installed and running as a Java JVM agent, which is configured as part of starting the JVM. The Camel application is running in the same JVM but not embedding Jolokia, as you did in the previous example. The Jolokia agent uses the HTTP server from Java to expose its services, which the web console accesses. The agent queries the JVM using JMX to discover the available JMX MBeans, which the agent can use to obtain the requested information from the web console.

The book's source code contains an example of using the Jolokia Java agent in the chapter16/jolokia directory. You can try the example with the following Maven goal:

```
mvn compile exec:exec
```

If you're not familiar with JVM agents, we suggest you take a look at the pom.xml file from this example, as it shows how to use Maven to execute a Java application with a JVM agent.

The example can also be run without Maven. First you need to use Maven to download the needed dependencies, which can be done once:

```
mvn dependency:copy-dependencies
```

Then you need to build the example:

```
mvn clean install
```

And finally, you can run the example using this rather long command:

```
java -javaagent:lib/jolokia-jvm-1.3.7-agent.jar -cp target/dependency/*:target/chapter16-jolokia-2.0.0.jar camelinaction.JolokiaMain
```

Notice how to specify the JVM agent on the command line, using the `-javaagent` argument.

TIP You can find more information about the Jolokia JVM agent from its website: <https://jolokia.org/agent/jvm.html>.

Jolokia is hot and awesome

We highly recommend Jolokia. It's an awesome library that makes JMX fun and easy to use again. A little-known fact is that the author of Jolokia, Roland Huß, is an avid chili fan, and therefore named the project after the strongest chili, Jolokia, also known as the ghost chili (https://en.wikipedia.org/wiki/Bhut_jolokia).

The fabric8 and Fuse team from Red Hat have been using Jolokia for a long time, and we provide Jolokia out of the box in many of our software projects. For example, JBoss Fuse and all Java-based Docker images come with Jolokia JVM agent installed.

Now that you've seen how to check the health of your applications, as well as dipped your toes into the waters of managing Camel applications, it's time to learn how to keep an eye on what your applications are doing.

16.3 Tracking application activity

Beyond monitoring an application's health, you need to ensure that it operates as expected. For example, if an application starts malfunctioning or has stopped entirely, it may harm business. There may also be business or security requirements to track particular services for compliance and auditing.

A Camel application used to integrate numerous systems may often be difficult to track because of its complexity. It may have inputs using a wide range of transports, and schedules that

trigger more inputs as well. Routes may be dynamic if you’re using content-based routing to direct messages to different destinations. And errors are occurring at all levels related to validation, security, and transport. Confronted with such complexity, how can you keep track of the behavior of your Camel applications?

You do that by tracking the traces that various activities leave behind. By configuring Camel to leave traces, you can get fairly good insight into what’s going on, both in real time and after the fact. Activity can be tracked using logs, whose verbosity can be configured to your needs. Camel also offers a notification system that you can use.

Let’s look at how to use log files and notifications to track activities.

16.3.1 USING LOG FILES

Monitoring tools can be tailored to look for patterns, such as error messages in logs, and they can use pattern matching to react appropriately, such as by raising an alert. Log files have been around for decades, so any monitoring tool should have good support for efficient log file scanning. Even if this solution sounds basic, it’s one used extensively in today’s IT world.

Log files are read not only by monitoring tools but also by people, such as operations, support, or engineering staff. That puts a burden on both Camel and your applications to produce enough evidence so that both humans and machines can diagnose the issues reported.

Camel offers four options for producing logs to track activities:

- *Using core logs*—Camel logs various types of information in its core logs. Major events and errors are reported by default.
- *Using custom logging*—You can use Camel’s logging infrastructure to output your own log entries. You can do this from different places, such as from the route using the log EIP or log component. You can also use regular logging from Java code

to output logs from your custom beans.

- *Using Tracer*—Tracer is used for tracing how and when a message is routed in Camel. Camel logs, at the INFO level, each and every step a message takes. Tracer offers a wealth of configuration options and features.
- *Using notifications*—Camel emits notifications that you can use to track activities in real time.

Let's look at these options in more detail.

16.3.2 USING CORE LOGS

Camel emits a lot of information at the DEBUG logging level and an incredible amount at the TRACE logging level. These levels are appropriate only for development, where the core logs provide great details for the developers.

In production, you'll want to use the INFO logging level, which generates a limited amount of data. At this level, you won't find information about activity for individual messages—for that, you need to use notifications or the Tracer, which we cover in section 16.3.4.

The core logs in production usage usually provide only limited details for tracking activity. Important lifecycle events such as the application being started or stopped are logged, as are any errors that occur during routing.

16.3.3 USING CUSTOM LOGGING

Custom logging is useful if you're required to keep an audit log. With custom logging, you're in full control of what gets logged.

In EIP terms, it's the Wire Tap pattern that tackles this problem. By tapping into an existing route, you can tap messages to an audit channel. This audit channel, which is often an internal queue (SEDA or VM transport), is then consumed by a dedicated audit service, which takes care of logging the messages.

USING WIRE TAP FOR CUSTOM LOGGING

Let's look at an example. At Rider Auto Parts, you're required to log any incoming orders. Figure 16.8 shows orders flowing in from CSV files that are then wiretapped to an audit service before moving on for further processing.

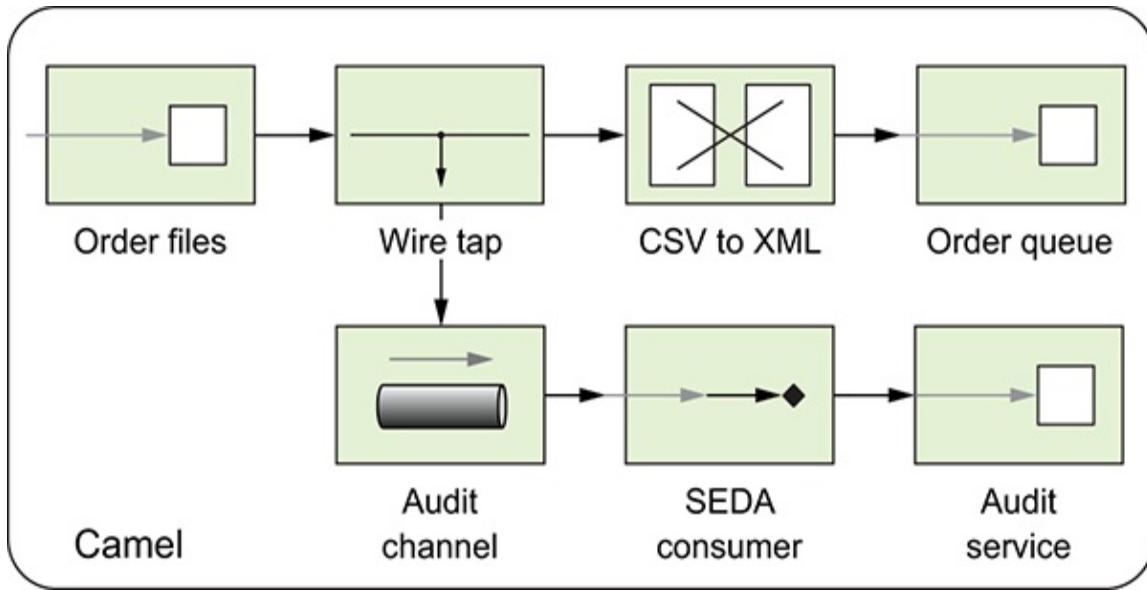


Figure 16.8 Using a wire tap to tap incoming files to an audit service before the file is translated to XML and sent to an order queue for further processing

Implementing the routes outlined in figure 16.8 in Camel is fairly straightforward, as shown in the following listing.

Listing 16.1 Using Wire Tap to tap messages for audit logging

```
public void configure() throws Exception {  
    from("file://rider/orders")  
        .wireTap("seda:audit")  
            ①
```

①

wireTap message to a seda queue

```
.bean(OrderCsvToXmlBean.class)  
.to("jms:queue:orders");  
  
from("seda:audit")
```

```
.bean(AuditService.class, "auditFile"); ②
```

②

Uses a bean to implement logic to write to audit log

```
}
```

The first route routes incoming order files. These are wiretapped to an internal SEDA queue ("seda:audit") ① for further processing. The messages are then transformed from CSV to XML using the OrderCsvToXmlBean bean before being sent to a JMS queue.

The second route is used for auditing. It consumes the tapped messages and processes them with an AuditService bean ②, as shown in the following listing.

[Listing 16.2](#) Implementation of a simple audit logging service using Java bean

```
public class AuditService {  
    private Log LOG =  
        LogFactory.getLog(AuditService.class); ①
```

①

Logger to use

```
    public void auditFile(String body) {  
        String[] parts = body.split(",");  
        String id = parts[0];  
        String customerId = parts[1];  
        String msg = "Customer " + customerId + " send  
order id " + id;  
        LOG.info(msg); ②
```

②

Logging an audit trail of the message

```
    }  
}
```

This implementation of the `AuditService` bean has been kept simple by logging the audit messages via a Java logger library ❶. The Java bean constructs a logging message using various parts of the message body, which then gets logged ❷.

NOTE The Wire Tap EIP uses a thread pool to process the tapped message concurrently. See more details in chapter 13, section 13.3.3.

The book's source code contains this example in the `chapter16/logging` directory. You can try the example using the following Maven goal:

```
mvn test -Dtest=AuditTest
```

USING THE CAMEL LOG COMPONENT

Camel provides a Log component that's capable of logging the Camel Message using a standard format at certain interesting points. To use the Log component, you route a message to it as follows:

```
public void configure() throws Exception {  
    from("file://rider/orders")  
        .to("log:input")  
        .bean(OrderCsvToXmlBean.class)  
        .to("log:asXml")  
        .to("jms:queue:orders");  
}
```

In this route, you use the Log component in two places. The first is to log the incoming file, and the second is after the transformation.

You can try this example using the following Maven goal from the `chapter16/logging` directory:

```
mvn test -Dtest=LogComponentTest
```

If you run the example, it will log the following:

```
2017-08-10 14:47:42,349 [et/rider/orders] INFO incoming -  
Exchange[ExchangePattern: InOnly, BodyType:  
org.apache.camel.component.file.GenericFile, Body: [Body is  
file based: GenericFile[ID-davsclaus-pro-55845-  
1439210860847-0-1]]]  
2017-08-10 14:47:42,351 [et/rider/orders] INFO asXml -  
Exchange[ExchangePattern: InOnly, BodyType: String, Body:  
<order><id>123</id><customerId>4444</customerId>  
<date>20170810</date>  
<item><id>222</id><amount>1</amount></item></order>]
```

By default, the Log component will log the message body and its type at the `INFO` logging level. Notice that in the first log line, the type is `GenericFile`, which represents a `java.io.File` in Camel. In the second log line, the type has been changed to `String`, because the message was transformed to a `String` using the `OrderCsvToXmlBean` bean.

You can customize what the Log component should log using the many options it supports. Consult the Camel Log documentation for the options (<http://camel.apache.org/log.html>). For example, to make the messages less verbose, you can disable showing the body type and limit the length of the message body being logged using the following configuration:

```
log:incoming?  
showExchangePattern=false&showBodyType=false&maxChars=100
```

That results in the following output:

```
2017-08-10 14:50:36,575 [et/rider/orders] INFO asXml -  
Exchange[Body: <order><id>123</id>  
<customerId>4444</customerId>  
<date>20170810</date><item><id>222</id><amount...>]
```

TIP The Log component has a `showAll` option to log everything from Exchange.

The Log component is used to log information from Exchange, but what if you want to log a message in a custom way? What you need is something like `System.out.println`, so you can input whatever `String` message you like into the log. That's where the Log EIP comes in.

USING THE LOG EIP

The Log EIP is built into the Camel DSL. It allows you to log a human-readable message from anywhere in a route, as if you were using `System.out.println`. It's primarily meant for developers, so they can quickly output a message to the log console. But that doesn't mean you can't use it for other purposes as well.

Suppose you want to log the filename you received as input. That's easy with the Log EIP—all you have to do is pass in the message as a String:

```
public void configure() throws Exception {  
    from("file://riders/orders")  
        .log("We got incoming file ${file:name} containing:  
${body}")  
        .bean(OrderCsvToXmlBean.class)  
        .to("jms:queue:orders");  
}
```

The String is based on Camel's Simple expression language, which supports the use of placeholders that are evaluated at runtime. In this example, the filename is represented by `${file:name}`, and the message body by `${body}`. If you want to know more about the Simple expression language, refer to appendix A.

You can run this example using the following Maven goal from the chapter16/logging directory:

```
mvn test -Dtest=LogEIPTest
```

If you run this example, it will log the following:

```
2017-08-10 14:52:32,576 [et/rider/orders] INFO route1 - We  
got incoming file someorder.csv containing:  
123,4444,20170810,222,1
```

The Log EIP will, by default, log at the `INFO` level using the route ID as the logger name. In this example, the route isn't explicitly named, so Camel assigns it the name `route1`.

Using the Log EIP from XML DSL is also easy, as shown here:

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
  <route>  
    <from uri="file://target/rider/orders"/>  
    <log message="Incoming file ${file:name} containing:  
${body}" />  
    <bean beanType="camelaction.OrderCsvToXmlBean"/>  
    <to uri="jms:queue:orders"/>  
  </route>  
</camelContext>
```

The XML example is also provided in the book's source code. You can try the example using the following Maven goal:

```
mvn test -Dtest=LogEIPSpringTest
```

The Log EIP also offers options to configure the logging level and log name, in case you want to customize those as well, as shown here in XML DSL:

```
<log message="Incoming file ${file:name} containing:  
${body}"  
  logName="Incoming" loggingLevel="DEBUG"/>
```

In the Java DSL, the logging level and log name are the first two parameters. The third parameter is the log message:

```
.log(LogLevel.DEBUG, "Incoming",  
  "Incoming file ${file:name} containing: ${body}")
```

Anyone who has had to browse millions of log lines to investigate an incident knows it can be hard to correlate messages.

USING CORRELATION IDs

When logging messages in a system, the messages being processed can easily get interleaved, which means the log lines will be interleaved as well. What you need is a way to correlate those log messages so you can tell which log lines are from which messages.

You do that by assigning a unique ID to each created message. In Camel, this ID is `ExchangeId`, which you can grab from `Exchange` using the `exchange.getId()` method.

TIP You can tell the Log component to log the `ExchangeId` using the following option: `showExchangeId=true`. When using the Log EIP, you can use `${id}` from the Simple expression language to grab the ID.

To help understand how and when messages are being routed, Camel offers Tracer, which logs message activity as it occurs.

16.3.4 USING TRACER

Tracer's role is to trace how and when messages are routed in Camel. It does this by intercepting each message being passed from one node to another during routing. [Figure 16.9](#) illustrates this principle.

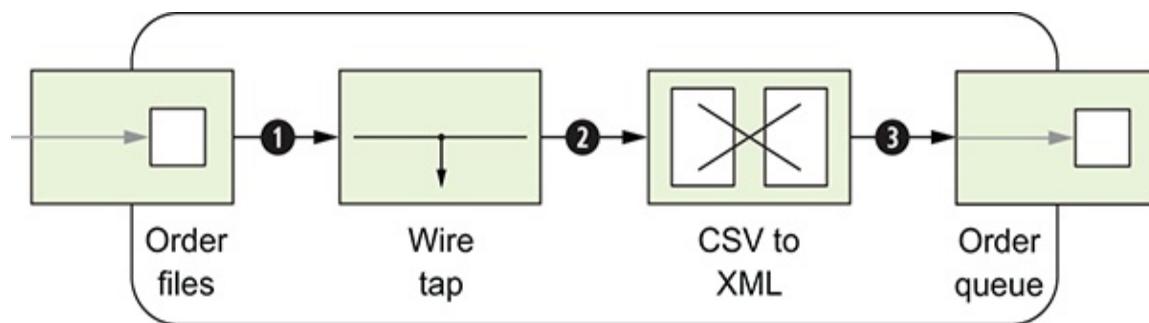


Figure 16.9 Tracer sits between each node in the route (at ①, ②, and ③) and traces the message flow.

You may remember being told that Camel has a channel sitting between each node in a route—at points ①, ②, and ③ in [figure 16.9](#). Channel1 has multiple purposes, such as error handling,

security, and interception. Because the Tracer is implemented as an interceptor, it falls under the control of channel, which at runtime will invoke it.

To use Tracer, you need to enable it, which is easily done in either the Java DSL or XML DSL. In the Java DSL, you can enable it by calling `context.setTracing(true)` from within the `RouteBuilder` class:

```
public void configure() throws Exception {  
    context.setTracing(true);  
    ...  
}
```

In XML DSL, you enable Tracer from `<camelContext>` as follows:

```
<camelContext trace="true"  
xmlns="http://camel.apache.org/schema/spring">
```

When running with Tracer enabled, Camel records trace logs at the `INFO` level, which at first may seem a bit verbose. To reduce the verbosity, we've configured Tracer to not show properties and headers. Here's an example of Tracer output:

```
2017-08-13 15:37:04,072 [et/rider/orders] INFO Tracer -  
ID-davsclaus-pro-56412-1439473020118-0-202 >>>(route1)  
from(file://target/rider/orders) -->  
wireTap(Endpoint[seda://audit]) <<<  
Pattern:InOnly,BodyType:org.apache.camel.component.file.GenericFile, Body:[Body is filebased: GenericFile[ID-  
davsclaus-pro-56347-1439472468659-0-117]]  
  
2017-08-13 15:37:06,076 [et/rider/orders] INFO Tracer -  
ID-davsclaus-pro-56412-1439473020118-0-202 >>>(route1)  
wireTap(Endpoint[seda://audit]) -->  
bean[camelinaction.OrderCsvToXmlBean@8801cab] <<<  
Pattern:InOnly,BodyType:org.apache.camel.component.file.GenericFile, Body:[Body is filebased: GenericFile[ID-  
davsclaus-pro-56347-1439472468659-0-117]]  
  
2017-08-13 15:37:06,076 [ - seda://audit] INFO Tracer -  
ID-davsclaus-pro-56412-1439473020118-0-205 >>>(route2)  
from(seda://audit) -->  
bean[camelinaction.AuditService@45aba779]<<<  
Pattern:InOnly,
```

```
BodyType:org.apache.camel.component.file.GenericFile, Body:  
[Body is file based: GenericFile[ID-davsclaus-pro-56347-  
1439472468659-0-117]]
```

The interesting thing to note from the trace logs is that the log starts with the exchange ID, which you can use to correlate messages. In this example, two IDs (highlighted in bold) are in play: **ID-davsclaus-pro-56412-1439473020118-0-202** and **ID-davsclaus-pro-56412-1439473020118-0-205**. You may wonder why you have two IDs when there's only one incoming message. That's because the wire tap creates a copy of the incoming message, and the copied message will use a new exchange ID because it's being routed as a separate process.

Next, Tracer outputs the message's current route, followed by the `from --> to` nodes. This is probably the key information when using Tracer, because you can see each individual step the message takes in Camel.

Then Tracer logs the message exchange pattern, which is either `InOnly` or `Inout`. Finally, it logs the information from the Message, just as the Log component would do.

Distributed tracing

In this chapter, we're covering tracing Camel applications in a nonclustered environment. In a distributed system, you need something more powerful to help orchestrate and gather data from all the running services in the cluster. We can recommend looking at the following open source projects for distributed tracing: Zipkin, OpenTracing, Jaeger (Uber), and Hawkular. Apache Camel has components that support Zipkin and OpenTracing along with examples you can find at <https://github.com/apache/camel/tree/master/examples>.

Monitoring applications via the core logs, custom logging, and

Tracer is like looking into Camel’s internal journal after the fact. If the log files get big, it may feel like you’re looking for a needle in a haystack. Sometimes you may prefer to have Camel call you when particular events occur. That’s where the notification mechanism comes into play.

16.3.5 USING NOTIFICATIONS

For fine-grained tracking of activity, Camel’s management module offers notifiers for handling internal notifications. These notifications are generated when specific events occur inside Camel, such as when an instance starts or stops, when an exception has been caught, or when a message is created or completed. The notifiers subscribe to these events as listeners, and they react when an event is received.

Camel uses a pluggable architecture, allowing you to plug in and use your own notifier, which we cover later in this section. Camel provides the following notifiers out of the box:

- `LoggingEventNotifier`—A notifier for logging a text representation of the event using the SLF4J logging framework. This means you can use loggers, such as log4j, which has a broad range of appenders that can dispatch log messages to remote servers using UDP, TCP, JMS, SNMP, email, and so on.
- `PublishEventNotifier`—A notifier for dispatching the event to any kind of Camel endpoint. This allows you to use Camel transports to broadcast the message any way you want.
- `JmxNotificationEventNotifier`—A notifier for broadcasting the events as JMX notifications. For example, management and monitoring tooling can be used to subscribe to the notifications.

You’ll learn in the following sections how to set up and use an event notifier and how to build and use a custom notifier.

WARNING Because routing each exchange produces at least two (created and completed) notifications, you can be overloaded with thousands of notifications. That’s why you should always

filter out unwanted notifications. The `PublishEventNotifier` uses Camel to route the event message, which will potentially induce a second load on your system. That's why the notifier is configured by default to not generate new events during processing of events.

CONFIGURING AN EVENT NOTIFIER

Camel doesn't use event notifiers by default, so to use a notifier, you must configure it. This is done by setting the notifier instance you want to use on the `ManagementStrategy`. When using the Java DSL, this is done as shown here:

```
LoggingEventNotifier notifier = new LoggingEventNotifier();
notifier.setLogName("rider.EventLog");
notifier.setIgnoreCamelContextEvents(true);
notifier.setIgnoreRouteEvents(true);
notifier.setIgnoreServiceEvents(true);
context.getManagementStrategy().addEventNotifier(notifier);
```

First you create an instance of `LoggingEventNotifier`, because you're going to log the events using log4j. Then you set the log name you want to use. In this case, you're interested in only some of the events, so you ignore the ones you aren't interested in.

The configuration when using XML DSL is a bit different, because Camel will pick up the notifier automatically when it scans the registry for beans of type `EventNotifier` on startup. This means you just have to declare a bean, like this:

```
<bean id="eventLogger"
      class="org.apache.camel.management.LoggingEventNotifier">
    <property name="logName" value="rider.EventLog"/>
    <property name="ignoreCamelContextEvents"
      value="true"/>
    <property name="ignoreRouteEvents" value="true"/>
    <property name="ignoreServiceEvents" value="true"/>
</bean>
```

You can also write your custom `EventNotifier` instead of using

the built-in notifiers.

USING A CUSTOM EVENT NOTIFIER

Rider Auto Parts wants to integrate an existing Camel application with the company's centralized error log database. They already have a Java library that's capable of publishing to the database, and this makes the task much easier. [Figure 16.10](#) illustrates the situation.

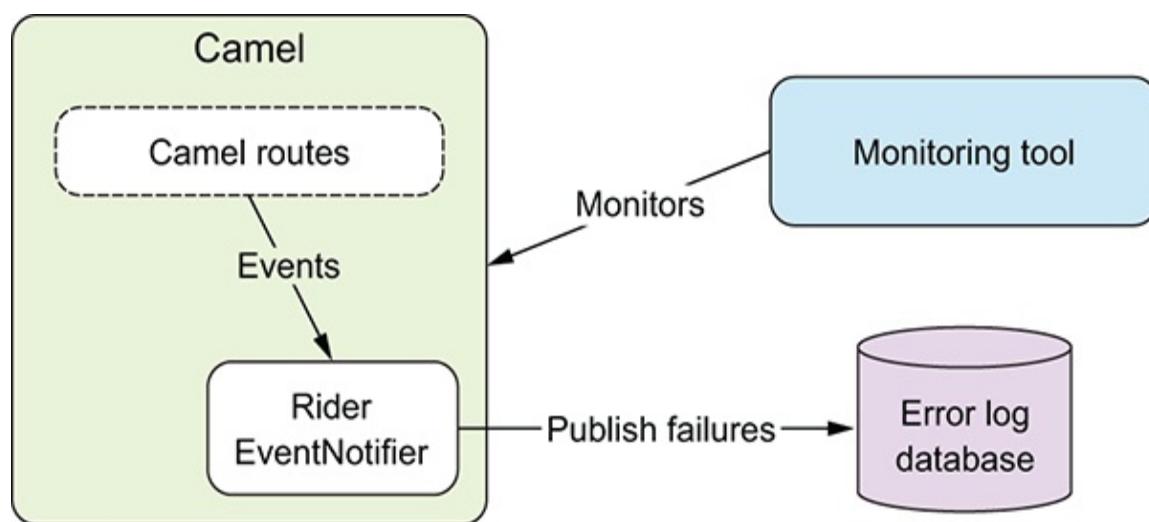


Figure 16.10 Failure events must be published into the centralized error log database using the custom `RiderEventNotifier`.

They decide to implement a custom event notifier named `RiderEventNotifier`, which uses its own Java code, allowing ultimate flexibility. [Listing 16.3](#) shows the important snippets for implementing this. Here, you extend the `EventNotifierSupport` class, an abstract class meant to be extended by custom notifiers. If you don't want to extend this class, you can implement the `EventNotifier` interface instead. The `RiderFailurePublisher` class is the existing Java library for publishing failure events to the database.

Listing 16.3 A custom event notifier publishes failure events to a central log database

```
public class RiderEventNotifier extends
```

```
EventNotifierSupport {  
    private RiderFailurePublisher publisher;  
  
    public boolean isEnabled(EventObject eventObject) {  
        return eventObject instanceof  
ExchangeFailedEvent; ①
```

①

Filters which events to trigger upon

```
}  
  
public void notify(EventObject eventObject) throws  
Exception { ②
```

②

Accepted events

```
if (eventObject instanceof ExchangeFailedEvent) {  
    ExchangeFailedEvent event =  
(ExchangeFailedEvent) eventObject;  
    String id =  
event.getExchange().getExchangeId();  
    Exception cause =  
event.getExchange().getException();  
    Date now = new Date();  
  
    publisher.publish(appId, id, now,  
cause.getMessage()); ③
```

③

Publishes failure events

```
}  
}  
  
protected void doStart() throws Exception {}  
  
protected void doStop() throws Exception {}  
}
```

The `isEnabled` method is invoked by Camel with the event being passed in as a `java.util.EventObject` instance. You use an

`instanceof` test to filter for the events you’re interested in, which are failure events ❶ in this example. If the `isEnabled` method returns `true`, the event is propagated to the `notify` method ❷. Then information is extracted from the event, such as the unique exchange ID and the exception message to be published. This information is then published using the existing Java library ❸.

TIP If you have any resources that must be initialized, Camel offers `doStart` and `doStop` methods for this kind of work, as shown in [listing 16.3](#).

The book’s source code contains this example in the `chapter16/notifier` directory, which you can try using the following Maven goal:

```
mvn test -Dtest=RiderEventNotifierTest
```

We’ve now reviewed four ways to monitor Camel applications. You learned to use Camel’s standard logging capabilities and to roll a custom solution when needed. In the next section, you take a further look at how to manage both Camel and your custom Camel components.

16.4 Managing Camel applications

Section 16.2 already touched on how to manage Camel, as you learned how to use JMX with Camel. This section takes deep dives into various management-related topics:

- *Camel application lifecycle*—Control your Camel application, such as stopping and starting routes, and much more using a broad range of ways with JMX, Jolokia, hawtio, and the `ControlBus` component
- *Camel management API*—Learn about the programming API from Camel that defines the management API.
- *Performance statistics*—Discover which key metrics Camel

captures about your Camel application performance, and how to access these metrics for custom reporting and hook into monitoring and alert tools.

- *Management enabling custom components*—Learn to program your custom Camel components and Java beans so they’re management enabled out of the box, as if they were first-class from the Camel release.

We’ll start by looking at how to manage the lifecycles of your Camel applications.

16.4.1 MANAGING CAMEL APPLICATION LIFECYCLES

Being able to manage the lifecycles of your Camel applications is essential. You should be able to stop and start Camel in a reliable manner, and you should be able to pause or stop a Camel route temporarily, to avoid taking in new messages while an incident is being mitigated. Camel offers you full lifecycle management on all levels.

Suppose you want to stop an existing Camel route. To do that, you connect to the application with JMX as you learned in section 16.2. [Figure 16.11](#) shows JConsole with the route in question selected.

As you can see, `route1` has been selected from the MBeans tree. You can view its attributes, which reveal various stats such as the number of exchanges completed and failed, its performance, and so on. The state attribute displays information about the lifecycle—whether it’s started, stopped, suspended, or resumed.

To stop the route, select the Operations entry and click the stop operation. Then return to the Attributes entry. You should see the state attribute’s value change to `Stopped`.

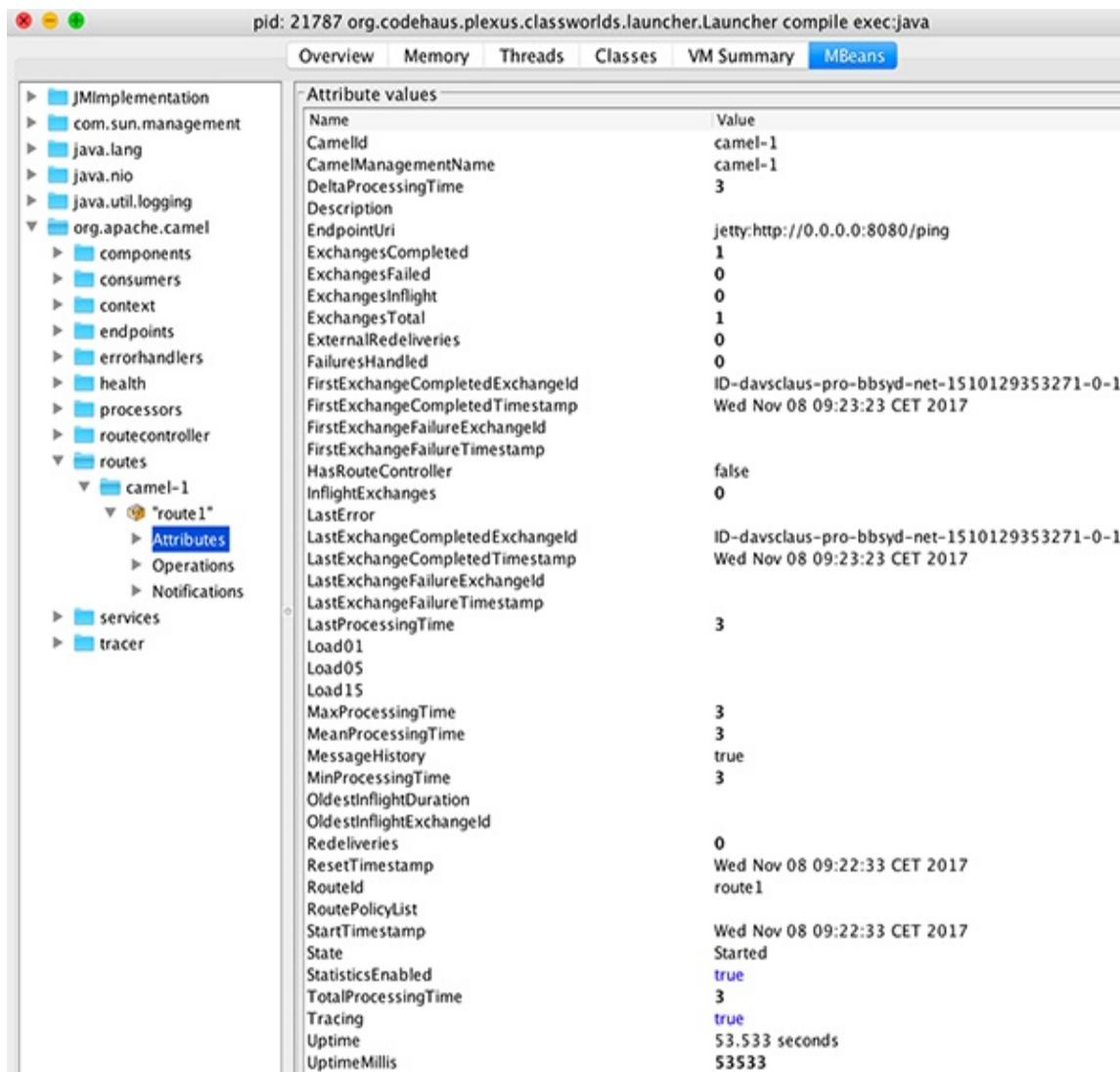


Figure 16.11 Selecting the route to manage in JConsole

TIP In JConsole, you can hover the mouse over an attribute or operation to show a tooltip with a short description.

You can also manage the lifecycle using Jolokia and hawtio.

16.4.2 USING JOLOKIA AND HAWTIO TO MANAGE CAMEL LIFECYCLES

Jolokia allows you to invoke JMX operations that you can use to start and stop Camel routes. We use an example from the source

code to demonstrate this. In the chapter16/jolokia-embedded-war file, run the following Maven goal:

```
mvn clean install
```

Then copy the WAR file to a running Apache Tomcat in the webapps directory:

```
cp target/chapter16-jolokia-embedded-war.war /opt/apache-tomcat-8.5.23/webapps/
```

You can then stop the Camel route using the following REST call using curl or from a web browser:

```
curl 'http://localhost:8080/chapter16-jolokia-embedded-war/jolokia/exec/org.apache.camel:context=camel-1,type=routes,name=%22route1%22/stop()'
```

From the Apache Tomcat log, you should see that Camel reports that it's stopping the route:

```
2017-08-19 11:05:20,700 [#0 - timer://foo] INFO  route 1 - I am running.  
2017-08-19 11:05:22,445 [nio-8080-exec-1] INFO DefaultShutdownStrategy - Starting to graceful shutdown 1 routes (timeout 300 seconds)
```

And likewise, you can start the route again using the start operation:

```
curl 'http://localhost:8080/chapter16-jolokia-embedded-war/jolokia/exec/org.apache.camel:context=camel-1,type=routes,name=%22route1%22/start()'
```

Now, you may have noticed that the REST call to Jolokia is a rather long command. How do you know this command? You use hawtio, which can display the Jolokia REST call for each JMX attribute or operation. Download hawtio and install it into the running Apache Tomcat, such as by copying the WAR file:

```
cp ~/Downloads/hawtio-default-1.5.6.war /opt/apache-tomcat-8.5.23/webapps/hawtio.war
```

Then access hawtio from a web browser:
<http://localhost:8080/hawtio>.

The JMX tab in hawtio is similar to JConsole, with a JMX tree on the left side. In the tree, you can find the Camel application and select the route MBean. And then select the Operations sub tab, to list the operations on this MBean. Find the stop operation and click it. This should lead to the screen shown in [figure 16.12](#).

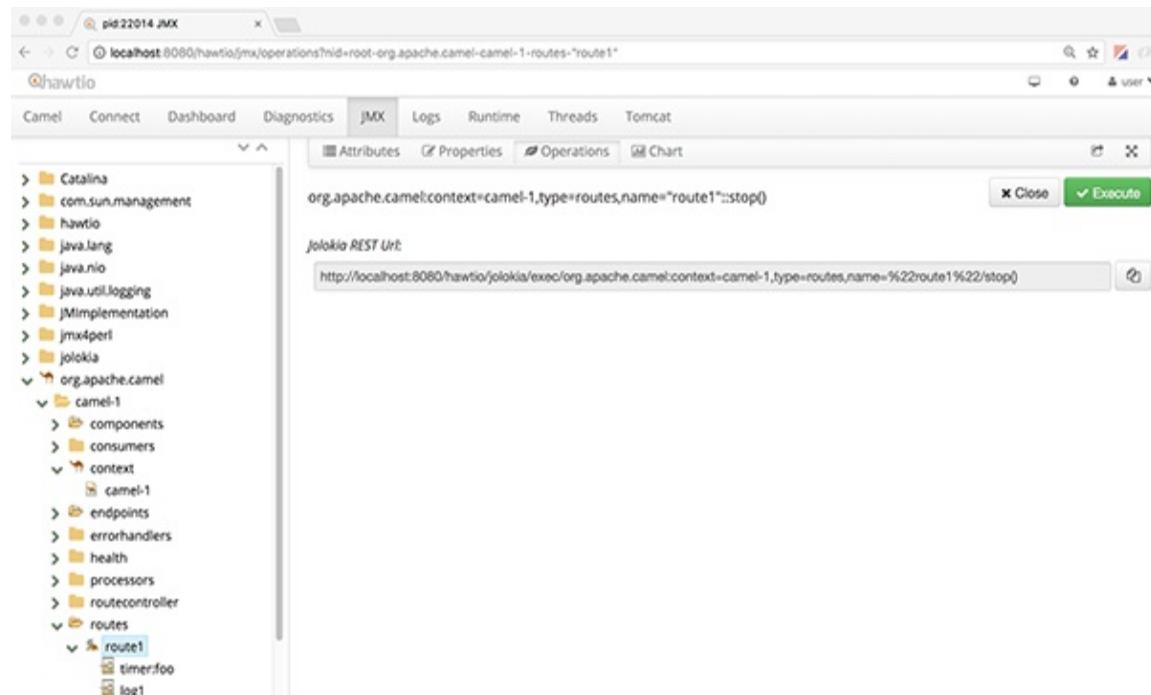


Figure 16.12 Using hawtio to manage a Camel application. In the JMX tab, the Camel route MBean is selected in the tree, and the stop operation is selected that allows you to invoke the operation using the Execute button. The corresponding Jolokia REST URL is also shown.

As [figure 16.12](#) shows, you can find every Jolokia REST URL using the hawtio web console.

TIP To manage Camel routes with hawtio, the Camel tab is more useful than the JMX tab. The Camel tab displays a list of all the routes in a table. You can easily select one or more routes to start or stop with a click of a button.

The Control Bus EIP pattern from the EIP book is the next topic.

16.4.3 USING CONTROL BUS TO MANAGE CAMEL

So far, the various ways for managing and controlling your Camel applications have been from the *outside*, but what if you want to do this from the *inside*? That's what the Control Bus EIP does, to monitor and manage the Camel application from within the framework.

In Camel, the Control Bus EIP is implemented as a Camel component that accepts commands to control the lifecycle of the Camel application. For example, you can send a message to a `controlbus` endpoint to stop a route or gather performance statistics.

Let's illustrate how this works with the example in the following listing.

Listing 16.4 The ping service implemented with Rest DSL and using Control Bus to control the route lifecycle

```
public class PingService extends RouteBuilder {  
    public void configure() throws Exception {  
  
        restConfiguration().component("restlet").port(8080); 1  
  
        rest("/rest").consumes("application/text").produces("application/text")  
            .get("ping") 2  
    }  
}
```

1

Uses restlet component on port 8080 as rest server

2

Specifies the ping service

```
.route().routeId("ping")
    .transform(constant("PONG\n"))
.endRest()

.get("route/{action}")
    .toD("controlbus:route?
routeId=ping&action=${header.action}"); ③
```

③

Control bus to control the route

```
}
```

When using the Camel Rest DSL (covered in chapter 10), you need to configure which Camel component to use as the HTTP REST-Server; in this example, we use the restlet component **①**. The ping service is an HTTP GET operation that returns the pong response **②**. The following rest service uses the context-path route/{action}, where action is a dynamic value that's bound as a Camel header, and therefore you're using dynamic-to as the controlbus endpoint. This allows you to specify the action parameter using the dynamic value of the header **③**.

The book's source code contains this example in the chapter16/controlbus directory. You can try this example by running the following Maven command:

```
mvn compile exec:java
```

Then from a web browser or using a tool like curl, you can control the route as shown here:

```
$ curl http://localhost:8080/rest/ping
PONG
$ curl http://localhost:8080/rest/route/status
Started
$ curl http://localhost:8080/rest/route/stop
$ curl http://localhost:8080/rest/route/status
Stopped
```

This example uses three of the possible action options the control bus accepts. All the possible values supported by the action option are listed in table [16.2](#).

Table 16.2 Supported values of the action option from the ControlBus component

Value	Description
start	Starts the route. There's no return value.
stop	Stops the route. There's no return value.
suspen	Suspends the route. There's no return value.
resume	Resumes the route. There's no return value.
status	Gets the route status returned as a plain-text value such as started, and/or stopped.
stats	Gets the route performance status returned in XML format.

The control bus executes the action synchronously by default. For example, if a route takes 15 seconds to stop gracefully, that action has to complete before the Camel routing engine can continue routing the message. If you want to execute the task asynchronously, you can do that by setting the option `async` to `true`:

```
.toD("controlbus:route?  
routeId=ping&action=${header.action}&async=true")
```

So far, you've seen how to manage Camel applications using various tools such as JMX and the control-bus component. They all operate on top of the Camel management API, which we'll look at in the next section.

16.5 The Camel management API

The Camel management API is defined as a set of JMX MBean

interfaces in the `org.apache.camel.api.management.mbean` package. These interfaces declare the JMX operations and JMX attributes that each and every Camel MBean provides. This is done using annotations to declare whether it's a read-only or read-write attribute, or an operation. For example, `ManagedCamelContextMBean` is the management API of `CamelContext`. The following listing shows snippets of the source code for this interface.

Listing 16.5 Source code of ManagedCamelContextMBean

```
package org.apache.camel.api.management.mbean;

import org.apache.camel.api.management.ManagedAttribute;
import org.apache.camel.api.management.ManagedOperation;

public interface ManagedCamelContextMBean extends
    ManagedPerformanceCounterMBean {

    @ManagedAttribute(description = "Camel ID") ①
```

①

Exposes attribute for management (read-only on getter)

```
        String getCamelId();

    @ManagedAttribute(description = "Camel
ManagementName") ①
        String getManagementName();

    @ManagedAttribute(description = "Camel Version") ①
        String getCamelVersion();

    @ManagedOperation(description = "Starts all the
routes") ②
        void startAllRoutes() throws Exception;
```

②

Exposes operation for management

```
void startAllRoutes() throws Exception;
```

In the `ManagedCamelContextMBean` interface, each getter/setter becomes a JMX attribute by annotating the getter and setter methods with the `@ManagedAttribute` annotation ❶. The attribute can be read-only if there's only an annotation on the getter method. The attribute is read-write when both the getter and setter have been annotated. Operations are regular Java methods that may return a response or not (`void`). The operation ❷ in [listing 16.5](#) doesn't return a response and is therefore declared as `void`.

The Camel management API is vast, as we have MBeans for almost all the moving pieces that Camel uses at runtime. The MBeans are divided into categories, as previously listed in [table 16.1](#). For example, there are MBeans for all the Camel components, data formats, endpoints, routes, processors (EIPs), thread pools, and miscellaneous services. These MBeans are defined as interfaces in the `org.apache.camel.api.management.mbean` package.

In section 16.5.2, you'll see an example of using the Camel MBean API to get runtime information from a throttler EIP. The management API can be accessed using regular JMX Java code, and also from within Camel. We cover all these aspects in the following two sections.

16.5.1 ACCESSING THE CAMEL MANAGEMENT API USING JAVA

The Camel management API is defined as a set of JMX MBeans, which any Java client can access. This allows you to write Java code that can manage Camel applications (or any Java application that offers a JMX management API). You may want to do this as part of integrating Camel management from an existing Java management library or tool.

NOTE Using JMX to manage Java applications has some drawbacks. For example, the client must use Java, and the network transport uses Java RMI, which isn't firewall friendly,

as multiple ports must be open. See the sidebar “JMX: The good, bad, and ugly” in section 16.2.2 for more details.

In this section, you’ll use a simple example that runs a Camel application in a JVM. Then from another JVM, the JMX client connects to the Camel JVM and, using the JMX API, is able to obtain information about the Camel application. This scenario is illustrated in figure 16.13.

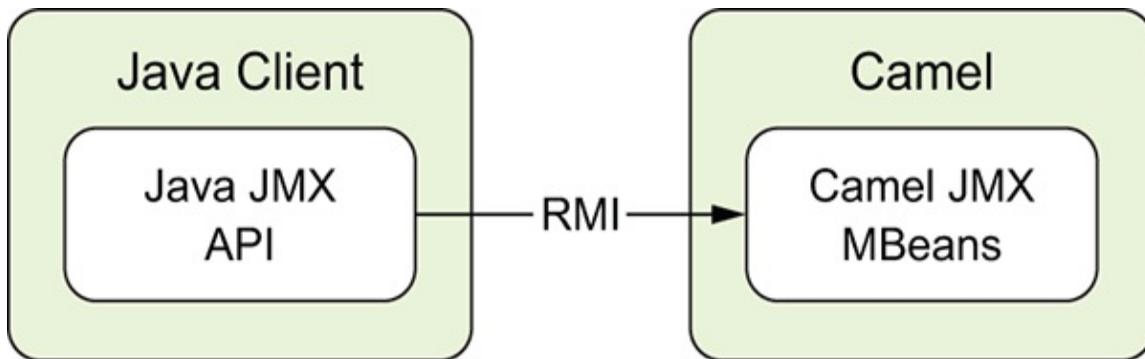


Figure 16.13 The Java client uses JMX API to connect and manage a remote Camel application, using the Camel JMX management API.

The example includes two clients. The first client uses solely the standard JMX API from the Java runtime. The second client uses a technique called a JMX proxy.

USING STANDARD JAVA JMX API

For a Java JMX client to be able to remotely manage another Java application, the remote application must expose a JMX connector that the client can connect and use. The Camel application does this by setting the `createConnector` option to `true`:

```
context.getManagementStrategy()  
    .getManagementAgent().setCreateConnector(true);
```

When Camel starts, a JMX connector is created and listens on port 1099 (the default port number). The URL that clients need to use for connecting to Camel is logged:

```
JMX Connector thread started and listening  
at:service:jmx:rmi:///jndi/rmi://davsclaus-  
pro:1099/jmxrmi/camel
```

This can be handy, as the URL isn't easy to remember.

The following listing shows how to write a Java application that connects to the remote Camel application and outputs the Camel version and the uptime.

[Listing 16.6](#) Java client connecting to a remote Camel application

```
public class JmxClientMain {  
    private String serviceUrl =  
"service:jmx:rmi:///jndi/rmi://" +  
"localhost:1099/jmxrmi/camel"; ①
```

JMX URL to connect to remote Camel application

```
public static void main(String[] args) throws Exception  
{  
    JmxClientMain main = new JmxClientMain();  
    main.run();  
}  
  
public void run() throws Exception {  
    JmxCamelClient client = new JmxCamelClient();  
    client.connect(serviceUrl); ②
```

Connecting to the Camel application

```
System.out.println("Version: " +  
client.getCamelVersion()); ③
```

Getting information about the Camel application

```
System.out.println("Uptime: " +  
client.getCamelUptime()); ③
```

```
client.disconnect(); 4
```

Disconnecting from the remote Camel application

```
}
```

There's more to this client, as we've implemented the lower-level JMX code in another class called `JmxCamelClient`, shown in the following listing.

Listing 16.7 Lower-level JMX API to remotely manage a Camel application

```
import javax.management.MBeanServerConnection; 1
```

1

Imports only standard Java JMX API

```
import javax.management.ObjectName; 1
import javax.management.remote.JMXConnector; 1
import javax.management.remote.JMXConnectorFactory; 1
import javax.management.remote.JMXServiceURL; 1

public class JmxCamelClient {
    private JMXConnector connector;
    private MBeanServerConnection connection;

    public void connect(String serviceUrl) throws Exception
{
```

```
    JMXServiceURL url = new
JMXServiceURL(serviceUrl); 2
```

2

Sets up URL to connect to the remote JVM

```
    connector = JMXConnectorFactory.connect(url,
null); 3
```

3

Connects to the remote JVM using the URL

```
connection =  
connector.getMBeanServerConnection(); ④
```

④

Opens a connection the client uses

```
}
```

```
public String getCamelVersion() throws Exception {  
    ObjectName on = new ObjectName("org.apache.camel:  
        + "context=camel-1,type=context,name=\"camel-  
1\""); ⑤
```

⑤

JMX ObjectName for the CamelContext

```
    return (String) connection.getAttribute(on,  
"CamelVersion"); ⑥
```

⑥

Reads the JMX attribute using the remote connection

```
}
```

```
public String getCamelUptime() throws Exception {  
    ObjectName on = new ObjectName("org.apache.camel:  
        + "context=camel-1,type=context,name=\"camel-  
1\""); ⑤
```

⑤

JMX ObjectName for the CamelContext

```
    return (String) connection.getAttribute(on,  
"Uptime"); ⑥
```

⑥

Reads the JMX attribute using the remote connection

```
}

public void disconnect() throws Exception {
    connector.close(); 7
```

7

Disconnects from the remote JVM

```
}
```

```
}
```

You can see from the top of [listing 16.7](#) that this class imports only the standard Java JMX API **1**. To connect to a remote JVM, three lines of code are needed (**2** **3** **4**) to obtain an `MBeanServiceConnection` instance, which is used to read JMX attributes (**5** **6**). When the client is done, it must close **7** so the connection is properly shut down.

The lower-level JMX code in [listing 16.7](#) can also use the Camel management API.

USING A JMX PROXY

When using a JMX proxy, the lower-level JMX code is hidden behind a Java interface, which acts as a facade, allowing the client to use the API from the interface as if it was a regular Java API. The Camel management API is defined in Java interfaces, and therefore JMX clients can use the JMX proxy technique with Camel.

The following listing shows how to use these interfaces to simplify the code.

Listing 16.8 Lower-level JMX API using JMX proxy as a facade for Camel management

```
import javax.management.JMX;
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
```

```
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

import
org.apache.camel.api.management.mbean.ManagedCamelContextMBean; ①
```

①

Imports the Camel management API

```
public class JmxCamelClient2 {

    private JMXConnector connector;
    private MBeanServerConnection connection;
    private ManagedCamelContextMBean proxy;

    public void connect(String serviceUrl) throws Exception
{
    JMXServiceURL url = new JMXServiceURL(serviceUrl);
```

Connects to the remote JVM

```
    connector = JMXConnectorFactory.connect(url, null);
    connection = connector.getMBeanServerConnection();

    // create an mbean proxy so we can use the type-
safe api
    ObjectName on = new ObjectName("org.apache.camel:"
        + "context=camel-1,type=context,name=\"camel-
1\"");
}
```

JMX ObjectName for the CamelContext

```
proxy = JMX.newMBeanProxy(connection, on,
```

Creates a JMX Proxy using ManagedCamelContextMBean as facade

```
    ManagedCamelContextMBean.class);
}

public void disconnect() throws Exception {
```

```
        connector.close();
    }

    public String getCamelVersion() throws Exception {
        return proxy.getCamelVersion();
```

Reads the JMX attributes using the facade

```
}
```

```
public String getCamelUptime() throws Exception {
    return proxy.getUptime();
```

Reads the JMX attributes using the facade

```
}
```

```
}
```

A difference between [listing 16.7](#) and [16.8](#) is that you're now using the Camel management API as the facade **①**. When doing this, you must add camel-core on the classpath. The benefit of using the facade is that the entire Camel management API is accessible, as if you were coding with Camel. Another benefit is type safety, so you don't have to worry about whether your JMX attributes and operations are defined correctly to avoid runtime errors when starting your application.

The difference becomes more apparent when comparing how [listings 16.7](#) and [16.8](#) have implemented the two methods `getCamelVersion` and `getCamelUptime`. [Listing 16.7](#) uses the clumsy JMX API, and [listing 16.8](#) uses type-safe Java method calls.

The book's source code contains this example in the `chapter16/jmx-client` directory. To run the example, you need to start the Camel application first using the following:

```
mvn compile exec:java -Pserver
```

Then you can run the clients:

```
mvn compile exec:java -Pclient
```

```
mvn compile exec:java -Pclient2
```

The Camel management API can also be used from within Camel applications, where you can use Java code to obtain any information from Camel—for example, to integrate with custom monitoring systems, or gather information for reporting.

16.5.2 USING CAMEL MANAGEMENT API FROM WITHIN CAMEL

Rider Auto Parts has a legacy order system that can handle only at most five orders per minute. To accommodate this, a Camel route has been put in front of the legacy system that throttles the messages:

```
from("seda:orders")
    .throttle(5).timePeriodMillis(60000).asyncDelayed().id("orderThrottler")
        .to("seda:legacy");
```

You've then been told to build a solution that can report to the central monitoring system the number of orders currently being throttled by Camel. How can you build such a solution?

With all the knowledge you've gained from this chapter, you're in safe hands. You've learned that Camel exposes a wealth of information at runtime. [Table 16.1](#) listed all the categories of information available. What you need to get is the runtime information from the Throttler EIP. As you know, the Camel management API is defined in the org.apache.camel.api.management.mbean package, so you take a look there, and discover the ManagedThrottlerMBean interface, which represents the Throttler EIP. This MBean has the information you need:

```
@ManagedAttribute(description = "Number of exchanges
currently throttled")
int getThrottledCount();
```

To access the information from the throttler, you need to use the Camel management API as a client from within Camel. You can

do this, as you learned previously, using the JMX and JMX proxy technique. But, hey, you're using Camel, so there's usually an easier way.

The easy way with Camel is using `camelContext` to quickly get an MBean that represents a given EIP from any of the Camel routes. This can be done using the `getManagedProcessor` method from `CamelContext`.

The following listing shows how to develop a Java class that can get the number of throttled messages.

Listing 16.9 Class that reports the number of current throttled messages

```
import org.apache.camel.CamelContext;
import org.apache.camel.CamelContextAware;
import
org.apache.camel.api.management.mbean.ManagedThrottlerMBean
;

public class RiderThrottlerReporter implements
CamelContextAware { ①
```

①

Class is CamelContextAware to have CamelContext injected

```
private CamelContext context;

public CamelContext getCamelContext() {
    return context;
}

public void setCamelContext(CamelContext camelContext)
{
    this.context = camelContext;
}

public long reportThrottler() {
    ManagedThrottlerMBean throttler =
        context.getManagedProcessor("orderThrottler",
        ManagedThrottlerMBean.class); ②
```

2

Gets the throttler MBean

```
return throttler.getThrottledCount(); 3
```

3

Returns the number of current throttled messages

```
}
```

This source code is a simple Java class that gets the CamelContext injected because it implements CamelContextAware ①. To get the current number of throttled messages, you use CamelContext to look up the throttler MBean ②, which has the ID orderThrottler in the Camel route. With the MBean, you can easily get the number of throttled messages ③.

The accompanying source code contains this example in the chapter16/jmx-camel directory. You can try the example using the following Maven goal:

```
mvn test -Dtest=RiderThrottlerTest
```

The throttler MBean exposes information about the Throttler EIP. Camel provides similar MBeans for all the other kinds of EIPs, such as Content-Based Router, Splitter, Aggregator, Recipient List, and so on. In addition, those MBeans have common information such as performance statistics.

16.5.3 PERFORMANCE STATISTICS

The Camel management API also exposes a lot of performance statistics, such as the average, minimum, and maximum processing time, and much more. This information is available on three levels:

- *CamelContext*—Aggregated statistics for all the Camel routes and processors

- *Route*—Aggregated statistics for the current route and its processors
- *Processor*—Statistics for each individual processor

The information that's exposed is defined in the interface org.apache.camel.api.management.mbean.ManagedPerformanceCounterMBean as JMX attributes. The following lists the most usable information:

```

@ManagedAttribute(description = "Number of completed
exchanges")
long getExchangesCompleted() throws Exception;

@ManagedAttribute(description = "Number of failed
exchanges")
long getExchangesFailed() throws Exception;

@ManagedAttribute(description = "Number of inflight
exchanges")
long getExchangesInflight() throws Exception;

@ManagedAttribute(description = "Min Processing Time
[milliseconds]")
long getMinProcessingTime() throws Exception;

@ManagedAttribute(description = "Mean Processing Time
[milliseconds]")
long getMeanProcessingTime() throws Exception;

@ManagedAttribute(description = "Max Processing Time
[milliseconds]")
long getMaxProcessingTime() throws Exception;

@ManagedAttribute(description = "Total Processing Time
[milliseconds]")
long getTotalProcessingTime() throws Exception;

@ManagedAttribute(description = "Last Processing Time
[milliseconds]")
long getLastProcessingTime() throws Exception;

```

All this information is available out of the box. For example, [figure 16.11](#) is a screenshot from JConsole with the CamelRouteMBean. On this screenshot, you can see some of the

performance statistics such as the min/mean/max values.

New information that was recently added to Camel has the oldest duration of all the current inflight messages. This indicates which of all the inflight messages is currently taking the most time:

```
@ManagedAttribute(description = "Oldest inflight exchange duration")
Long getOldestInflightDuration();
```

The book's source code contains a little example of obtaining this information every second and logging to the console. You can run this example, which is in the chapter 16/jmx-camel directory, using the following Maven goal:

```
mvn test -Dtest=OldestTest
```

The implementation is similar to what you've done in [listing 16.9](#). ManagedRouteMBean is retrieved from camelContext using the following code, in which myRoute is the route ID:

```
ManagedRouteMBean route =
context.getManagedRoute("myRoute",
ManagedRouteMBean.class);
```

TIP The performance statistics can be dumped as XML using the dumpStatsAsXml(boolean) method. If the boolean parameter is true, the statistics include extended information.

The performance statistics out of the box from Apache Camel are specific to Camel. A popular metrics library is Dropwizard metrics (formerly known as codehale metrics). You can use this library with Camel using the camel-metrics component.

USING CAMEL-METRICS FOR ROUTE PERFORMANCE STATISTICS

The camel-metrics component allows you to easily capture route performance statistics for all your Camel routes. The statistics

can then be reported to various monitoring systems, or you can expose the information in JSON format, from a service over HTTP or JMX transport.

Enabling camel-metrics on all your Camel routes is done by `org.apache.camel.spi.RoutePolicyFactory`, which creates a new `RoutePolicy` for all your routes. The camel-metrics component offers `MetricsRoutePolicyFactory`, which can be used in Java or XML DSL:

```
context.addRoutePolicyFactory(new  
MetricsRoutePolicyFactory());
```

And in XML DSL, you declare the factory as a `<bean>`, and it's automatically enabled:

```
<bean id="metricsFactory" class="org.apache.camel.component  
.metrics.routepolicy.MetricsRoutePolicyFactory"/>
```

This uses Dropwizard metrics to capture the amount of time the route takes to process each message. This is done by a Dropwizard timer that measures both the rate that a particular piece of code is called and the distribution of its duration.

You can access the statistics from Java or JMX, as shown in the following listing.

Listing 16.10 Accessing Dropwizard performance statics from Java code

```
MetricsRegistryService registryService =  
context.hasService(MetricsRegistryService.class); ①
```

①

Obtains the `MetricsRegistryService` from `CamelContext`

```
if (registryService != null) {  
    MetricRegistry registry =  
    registryService.getMetricsRegistry(); ②
```

2

Gets hold of Dropwizard MetricRegistry that holds the statistics

```
long count = registry.timer("camel-  
1:foo.responses").getCount(); ③
```

3

Gets the response timer for the foo route as count, mean, rate1, rate5, and rate15

```
double mean = registry.timer("camel-  
1:foo.responses").getMeanRate();  
double rate1 = registry.timer("camel-1:foo.responses")  
    .getOneMinuteRate();  
double rate5 = registry.timer("camel-1:foo.responses")  
    .getFiveMinuteRate();  
double rate15 = registry.timer("camel-1:foo.responses")  
    .getFifteenMinuteRate();  
log.info("count={}, mean={}, rate1={}, rate5={},  
rate15={}",  
        count, mean, rate1, rate5, rate15);  
}
```

This code is from an example from the book's source code, which is located in the chapter16/metrics directory.

To get hold of the Dropwizard metrics, you need first to get hold of MetricsRegistryService from the CamelContext ①. The MetricRegistry ② then gives access to all the performance statistics that Dropwizard has gathered. The camel-metrics component uses a timer for each route, using the naming pattern camelId:routeId.responses. The example contains two routes named foo and bar, hence the code snippet in [listing 16.10](#) gathers the statistics for the foo route.

You can also get all the statistics at once in JSON format. You can do that using JMX, as shown in the following listing.

[Listing 16.11](#) Using JMX to get Dropwizard performance statistics in JSON format

```
ObjectName on = new  
ObjectName("org.apache.camel:context=camel-1,"  
+  
"type=services, name=MetricsRegistryService");
```

①

1
JMX ObjectName to the MetricsRegistryService

```
MBeanServer server = context.getManagementStrategy()  
.getManagementAgent().getMBeanServer();
```

②

2
Gets hold of the MBeanServer

```
String json = (String) server.invoke(on,  
"dumpStatisticsAsJson",  
null, null);
```

Invokes the dumpStatisticsAsJson operation

This code is from the same example shown in [listing 16.10](#). To get the performance statistics in JSON from JMX, you need to get hold of the JMX MBean ObjectName ① that has the dumpStatisticsAsJson operation ②. JMX is what hawtio uses in the Camel plugin to show the Dropwizard performance statistics in a graphical chart, as shown in [figure 16.14](#).

You can try this example by running the following goal from Maven from the chapter16/metrics directory:

```
mvn compile hawtio:run
```

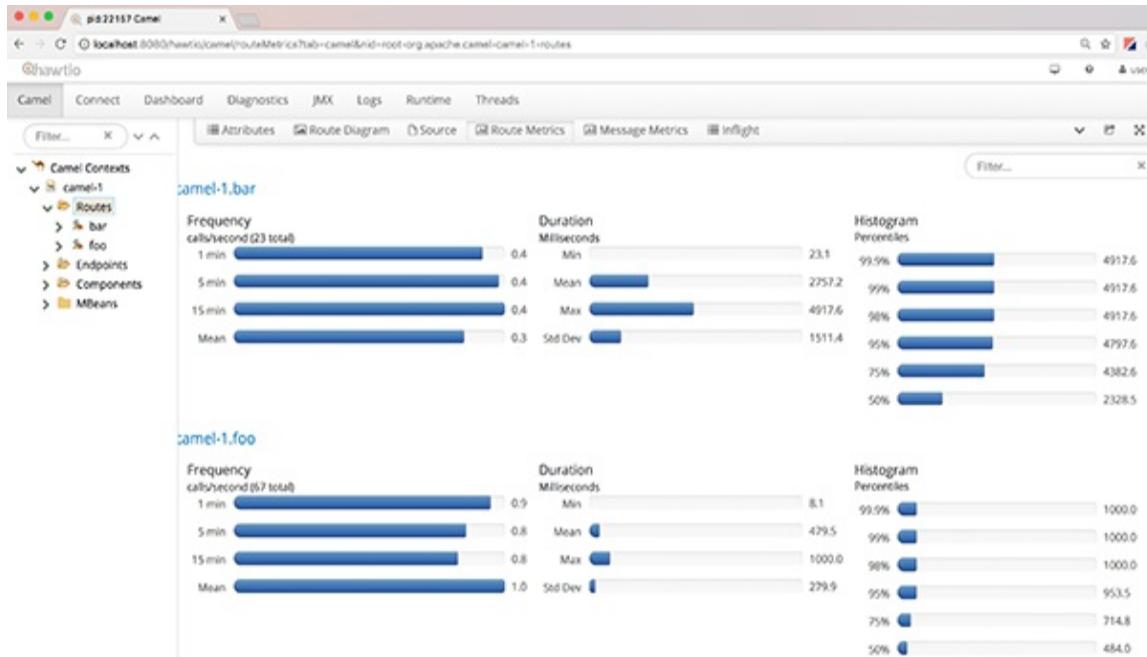


Figure 16.14 Hawtio web console showing Dropwizard performance statistics of each Camel route running in the application

And then you can find the Route Metrics tab in the Camel plugin and see the graphical chart in real time.

You may have built some Camel components of your own that you'd like to manage.

16.5.4 MANAGEMENT-ENABLE CUSTOM CAMEL COMPONENTS

Suppose Rider Auto Parts has developed a Camel ERP component to integrate with its ERP system, and the operations staff has requested that the component be managed. The component has a verbosity switch that should be exposed for management. Running with verbosity enabled allows the operations staff to retrieve additional information from the logs, which is needed when some sort of issue has occurred.

Listing 16.12 shows how to implement this on the `ERPEndpoint` class, which is part of the ERP component. This code listing has been abbreviated to show only the relevant parts of the listing; the full example is in the book's source code in the `chapter16/custom` directory.

Listing 16.12 Management enabling a custom endpoint

```
@ManagedResource(description = "Managed ERPEndpoint") 1
```

1

Exposes class as MBean

```
public class ERPEndpoint extends DefaultEndpoint {  
  
    private String name;  
    private boolean verbose;  
  
    public ERPEndpoint(String endpointUri, Component  
component) {  
        super(endpointUri, component);  
    }  
  
    @ManagedAttribute(description = "Verbose logging  
enabled")  
    public boolean isVerbose() { 2
```

2

Exposes attribute for management (read/write on getter and setter)

```
        return verbose;  
    }  
  
    @ManagedAttribute(description = "Verbose logging  
enabled")  
    public void setVerbose(boolean verbose) { 2
```

2

Exposes attribute for management (read/write on getter and setter)

```
        this.verbose = verbose;  
    }  
  
    @ManagedAttribute(description = "Logical name of  
endpoint")  
    public String getName() { 3
```

③

Expose attribute for management (read-only on getter)

```
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    @ManagedOperation(description = "Ping test of the ERP  
system")  
    public String ping() { ④
```

④

Expose operation for management

```
        return "PONG";  
    }  
  
}
```

If you've ever tried using the JMX API to expose the management capabilities of your custom beans, you know it's a painful API to use. It's better to go for the easy solution and use Camel JMX. You'll notice that it uses the Camel `@ManagedResource` annotation ① to expose this class as an MBean. In the same way, you can expose the `verbose` property as a managed read-write attribute using the `@ManagedAttribute` ② annotation on the setter method. Likewise, you can expose the `name` property as a managed read-only attribute when you annotate the getter method ③. JMX operations can also easily be exposed by annotation methods with `@ManagedOperation` ④.

You can run the following Maven goal from the `chapter16/custom` directory to try this example:

```
mvn compile exec:java
```

When you do, the console will output a log line every 5 seconds, as the following route illustrates:

```

from("timer:foo?period=5000")
    .setBody().simple("Hello ERP calling at
${date:now:HH:mm:ss}")
    .to("erp:foo")
    .to("log:reply");

```

What you want to do now is turn on the verbose switch from your custom ERP component. [Figure 16.15](#) shows how this is done from JConsole.

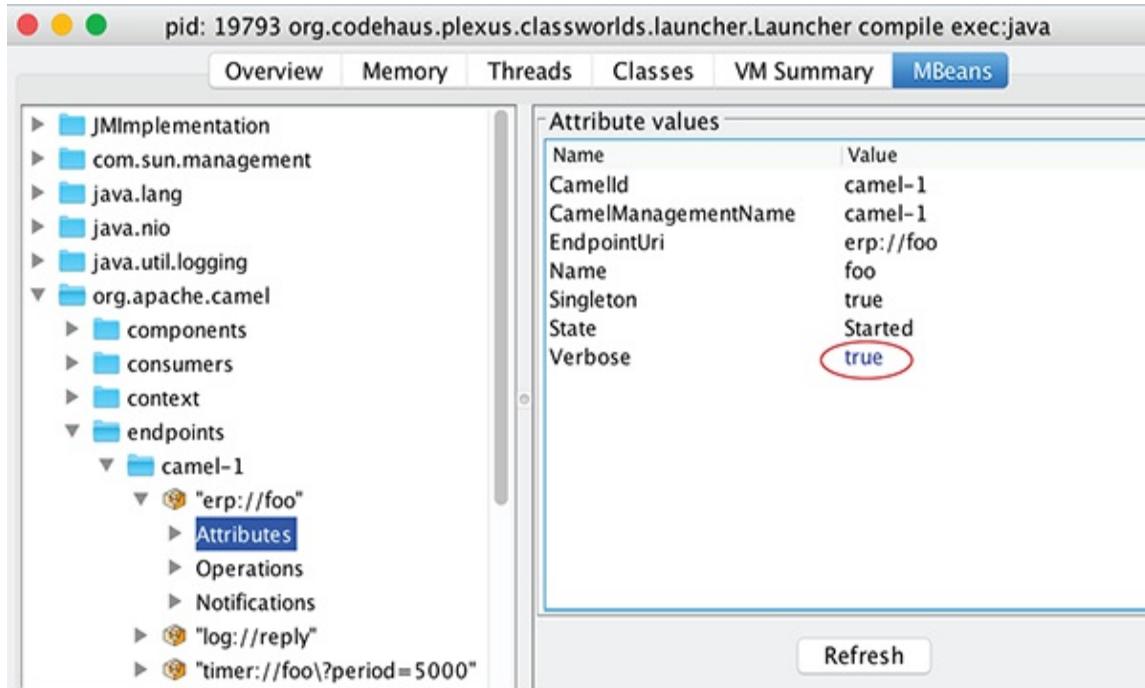


Figure 16.15 Enabling the `Verbose` attribute at runtime using JConsole. Click the Value column to edit and change the text from `false` to `true`.

As you can see, your custom component is listed under endpoints as `erp://foo`, which was the URI used in the route. The figure also shows the `Verbose` attribute. If you change this value to `true`, the console should immediately reflect this change. The first two of the following lines are from before the `verbose` switch was enabled. When the switch is enabled, it starts to output `Calling ERP...`:

```

2017-08-19 12:53:12,145 [0 - timer://foo] INFO reply -
Exchange[ExchangePattern: InOnly, BodyType:String, Body:
Simulated response from ERP]

```

```

2017-08-19 12:53:17,145 [0 - timer://foo] INFO reply -

```

```
Exchange[ExchangePattern: InOnly, BodyType:String, Body:  
Simulated response from ERP]
```

```
Calling ERP with: Hello ERP calling at 12:53:22
```

```
2017-08-19 12:53:22,145 [0 - timer://foo] INFO reply -  
Exchange[ExchangePattern: InOnly, BodyType:String, Body:  
Simulated response from ERP]
```

What you've just learned about management-enabling a custom component is the same principle Camel uses for its components. A Camel component consists of several classes, such as Component, Endpoint, Producer, and consumer, and you can management-enable any of those. For example, the schedule-based components, such as the Timer component, allow you to manage the consumers to adjust how often they should trigger.

It's not only custom Camel components that you can management-enable using the Camel annotations. You can also do this on regular Java beans (POJOs).

16.5.5 MANAGEMENT-ENABLE CUSTOM JAVA BEANS

To enable custom Java beans for management, you use the same approach as for custom Camel components: you use the Camel management annotations.

The following listing shows how a simple Java bean can be enabled for management.

Listing 16.13 The Java bean has been management-enabled using the Camel annotations

```
@ManagedResource
```

1

1
Exposes class as MBean

```
public class HelloBean {
```

```
private String greeting = "Hello";  
  
    @ManagedAttribute(description = "The greeting to  
use") ②
```

②

Exposes attribute for management (read/write on getter and setter)

```
public String getGreeting() {  
    return greeting;  
}  
  
    @ManagedAttribute(description = "The greeting to  
use") ②
```

②

Exposes attribute for management (read/write on getter and setter)

```
public void setGreeting(String greeting) {  
    this.greeting = greeting;  
}  
  
    @ManagedOperation(description = "Say the  
greeting") ③
```

③

Exposes operation for management

```
public String say() {  
    return greeting;  
}
```

By adding the Camel management annotations to the bean, you can expose it as an MBean. This is done by adding `@ManagedResource` on the class level ①. Attributes on the bean can be exposed using `@ManagedAttribute` ②, which supports read-only and read-write mode. In this example, you add the `@ManagedAttribute` on both the getter and setter, hence the attribute is read-write. For read-only mode, the annotation should be configured only on the getter. A JMX operation can be

exposed using `@ManagedOperation` ❸, which in this example invokes the `say` method.

The bean must be used in a Camel route to let Camel enlist the bean as a Camel processor in JMX. This process happens when Camel is starting and all the routes are built from the model, and part of that process is to enlist MBeans that represent the various parts of the routes such as consumers, producers, endpoints, EIPs, and so on. These MBeans are categorized as shown in table 16.1.

The book's source code contains an example with the `hello` bean in the `chapter16/custom-bean` directory. In this example, the bean is used in a simple Camel route:

```
from("timer:foo?period=5s")
    .bean("hello", "say")
    .log("${body}");
```

The application uses a Camel `Main` class to boot Camel. As part of the configuring of this class, a `HelloBean` instance is created and enlisted in the Camel registry using the name `hello` ❶:

```
public static void main(String[] args) throws Exception {
    Main main = new Main();
    main.bind("hello", new HelloBean());
```

❶

Binds a `HelloBean` instance in the Camel registry

```
    main.addRouteBuilder(new HelloRoute());
    main.run();
}
```

You can try this example by running the following Maven goal from the `chapter16/custom-bean` directory:

```
mvn compile exec:java
```

If you run the example and connect to the JVM using JConsole, you can find the custom bean in the JMX tree under the Camel processor tree, as shown in figure 16.16.

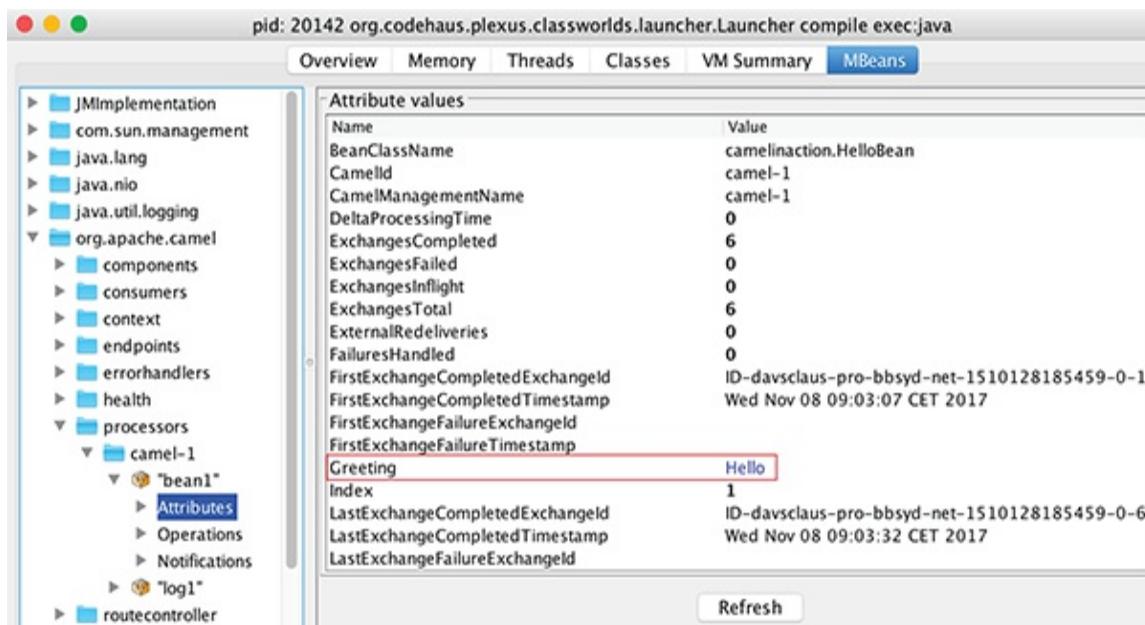


Figure 16.16 HelloBean with the custom Greeting attribute is enlisted in the Camel processor tree, also inheriting the standard set of Camel JMX attributes such as all the performance details.

Congratulations! You've now learned all there is to managing Camel applications and enlisting your custom components or Java beans for management.

16.6 Summary and best practices

A sound strategy for monitoring your applications is necessary when you take them into production. Your organization may already have strategies that must be followed for running and monitoring applications.

In this chapter, you looked at monitoring your Camel applications using health-level checks. You learned that existing monitoring tools could be used via SNMP, JMX protocols, or REST with help from Jolokia. Using JMX allows you to manage Camel at the application level, which is essential for lifecycle management, and performing functions such as stopping a route. You also looked at what Camel has to offer in terms of logging. You learned about the Camel logs and using custom logging. You also explored the Camel notification system, which is pluggable, allowing you to hook in your own notification

adapter and send notifications to a third party. The last section of this chapter covered the Camel management API. We explored how you can access, from this API, details such as performance statistics. We also discussed how you can build custom Camel components or Java beans that can tap into the Camel management API as a first-class citizen.

Here are a few simple guidelines:

- *Involve the operations team*—Monitoring and management aren’t afterthoughts. You should involve the operations team early in the project’s lifecycle. Your organization likely already has procedures for managing applications, which must be followed.
- *Use health checks*—For example, develop a *happy* page that does an internal health check and reports back on the status. A happy page can then easily be accessed from a web browser and monitoring tools.
- *Provide informative error messages*—When something goes wrong, you want the operations staff receiving the alert to be able to understand what the error is all about. If you throw exceptions from business logic, include descriptive information about what’s wrong.
- *Use the Tracer*—If messages aren’t being routed as expected, you can enable Tracer to see how they’re being routed. But beware; Tracer can be verbose, and your logs can quickly fill up with lines if your application processes a lot of messages.
- *Read log files from testing*—Have developers read the log files to see which exceptions have been logged. This can help them preemptively fix issues that otherwise could slip into production.
- *Jolokia is awesome*—Jolokia brings fun back to Java management. You can now easily enable your custom Camel components and Java beans for management and let clients easily access and manage them with the help from Jolokia, using REST and HTTP.

- *Distributed tracings*—Users building distributed applications or a microservice-based architecture should consider a strategy for tracing applications. Unfortunately, this topic came too late to be covered in this book. We recommend looking at OpenTracing or Zipkin and what's happening with distributed tracing in cloud platforms, because such a functionality is likely to be a must-have feature in the near future. You can find examples of using both Zipkin and OpenTracing from Apache Camel at <https://github.com/apache/camel/tree/master/examples>.

The next chapter covers how to cluster your Camel applications. You'll learn how to set up Camel in a highly available cluster, how to cluster your Camel routes, and how to use clustered messaging with Camel. Another way to say it: it's clustering time!

Part 6

Out in the wild

The last part of the book is where we go wild.

The first and most important topic is about how to cluster your Camel applications. In chapter 17, you'll find a range of different examples of how to cluster with HTTP, files, messaging systems, distributed caches, and schedulers. That's not all—you can also cluster by container-based infrastructure, which is covered in the following chapter.

Chapter 18 takes you on a wild ride by first covering how to get started using Camel in the world of Docker and Kubernetes container platforms. Docker and Kubernetes have gained rapid popularity and are becoming the de facto standard for running applications in the cloud.

Chapter 19 walks you through the various Camel tooling. You'll learn about the Camel Maven plugin that comes out of the box with the Apache Camel release, and we'll do a breakdown of the most interesting third-party Camel tooling on the internet. Appendix A discusses the deceptively named Simple language, which has become quite powerful. And finally, appendix B lists various useful resources where you can find more information about Apache Camel in the wider Camel community.

That's it for the book—no, wait, actually we're *still* not done. You'll find two additional bonus chapters online at www.manning.com/books/camel-in-action-second-edition. Chapter 20 gives an introduction to reactive systems and the Reactive Streaming API that you can use together with Camel. The second half of the chapter is a brief introduction to a reactive

framework called Vert.x. It shows you how to use Vert.x and Camel together to build reactive systems.

For the second bonus chapter, Chapter 21, we invited a guest author, Henryk Konsek, to introduce you to IoT (the Internet of Things) and explain how you can make Camel part of the IoT world.

17

Clustering

This chapters covers

- Clustered HTTP
- Clustered Camel routes in active/passive and active/active modes
- Clustered messaging with JMS and Kafka
- Clustered caches
- Clustered scheduling
- Calling clustered services using the Service Call EIP

“How do I set up a highly available cluster with Camel?” That’s a simple question that doesn’t have a straightforward answer. You can ask the same question about a Java application server such as Apache Tomcat or WildFly and get a well-documented answer. Unlike Camel, these application servers are within a well-defined scope. They often support only a few protocols such as HTTP (Servlet) and messaging (JMS). Camel, on the other hand, speaks a lot more protocols, as you can tell by the many Camel components.

For example, a Camel application may expose a REST service and place its content into a database or filesystem. Or it may accept HL7 messages over TCP and route to a JMS broker. Or you may use Camel to stream from Kafka topics to a cloud

service running on AWS. The answer to the question “How do I set up a highly available cluster with Camel?” depends on what you do with Camel.

To understand clustering support in Camel, we have to go back to the beginning. Camel was created in 2007 as a lightweight integration framework. Back then, Camel was just a set of JARs that you added to your classpath, so you could then run Camel in any JVM. Typically, you’d embed Camel into an existing application server or ESB.

At that time, any clustering was beyond the scope of Camel’s core module. *Clustering* was intended to come from a third-party (such as application servers and ESBs) or from the Camel components that facilitate clustering.

Historically, Camel came from the ESB world—from Apache ServiceMix. ServiceMix would then bring support for clustering to its service bus. That clustering was based on messaging and came from Apache ActiveMQ.

Fast-forward to today, and the landscape has changed. Clustering in Camel is mainly driven by the following:

- *Camel components*—Camel components with native clustering support
- *Clustered route policies*—Running Camel routes in active/passive mode (also called *master/slave*)
- *Container-based infrastructure*—Clustering by the infrastructure (covered in chapter 18)

This chapter covers various use cases illustrating how to cluster Camel with some of the most common protocols such as HTTP, files, JMS, Kafka, and clustered caches. We’ll also show you how to set up clustered Camel routes in master/slave mode, ensuring that only one route is active at any time. The last section covers the Service Call EIP, which is used for calling services in a clustered and distributed system.

Let’s start with one of the simplest clustering techniques:

HTTP clustering using load balancing.

17.1 Clustered HTTP

A dozen or so of the Camel components speak HTTP—such as CXF, CXF-RS, Jetty, Undertow, and Servlet, to name a few. These components can all be used in Camel to define HTTP-based services. And because HTTP is prolific, it's probably the most easily understood clustering scenario we'll cover.

HTTP has two modes: stateless and stateful. *Stateless* is by far the most commonly used and is our recommended approach. It scales well, requires no coordination between the nodes in the cluster, and is much easier to set up. You run as many instances of the HTTP service as you need, and the services can be colocated or spread across data centers. The clustering isn't provided by Camel or the HTTP service, but is done with the help of a load balancer in front of each HTTP service, as illustrated in figure 17.1.

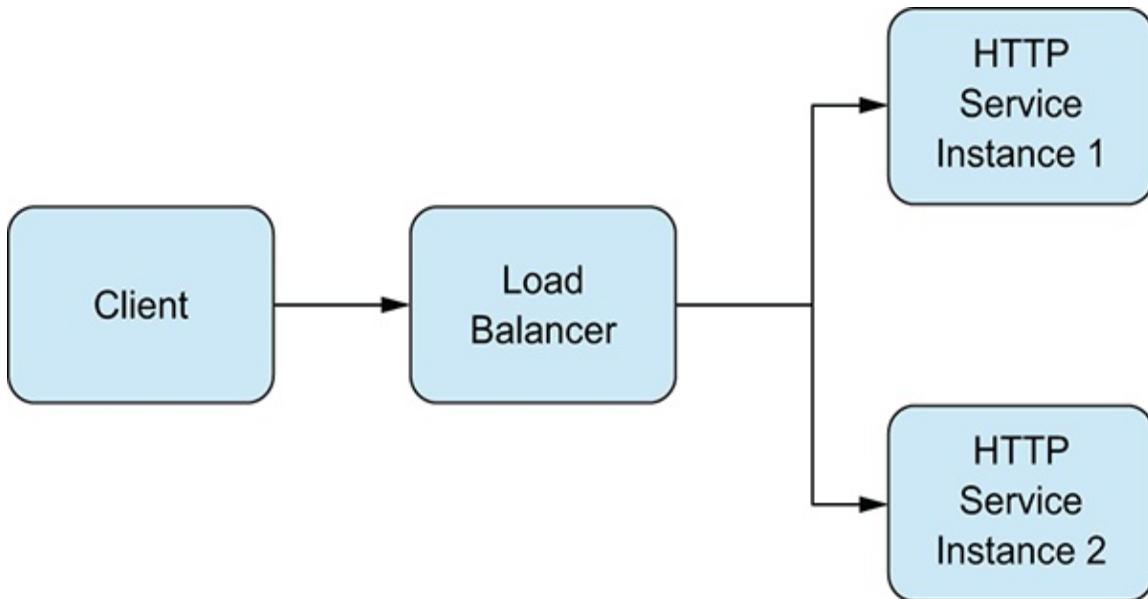


Figure 17.1 The load balancer fronts the HTTP services and directs traffic from clients to available instances in the cluster.

If any instances of the HTTP service goes down, the load balancer will redirect traffic to one of the other available

instances, as illustrated in [figure 17.2](#).

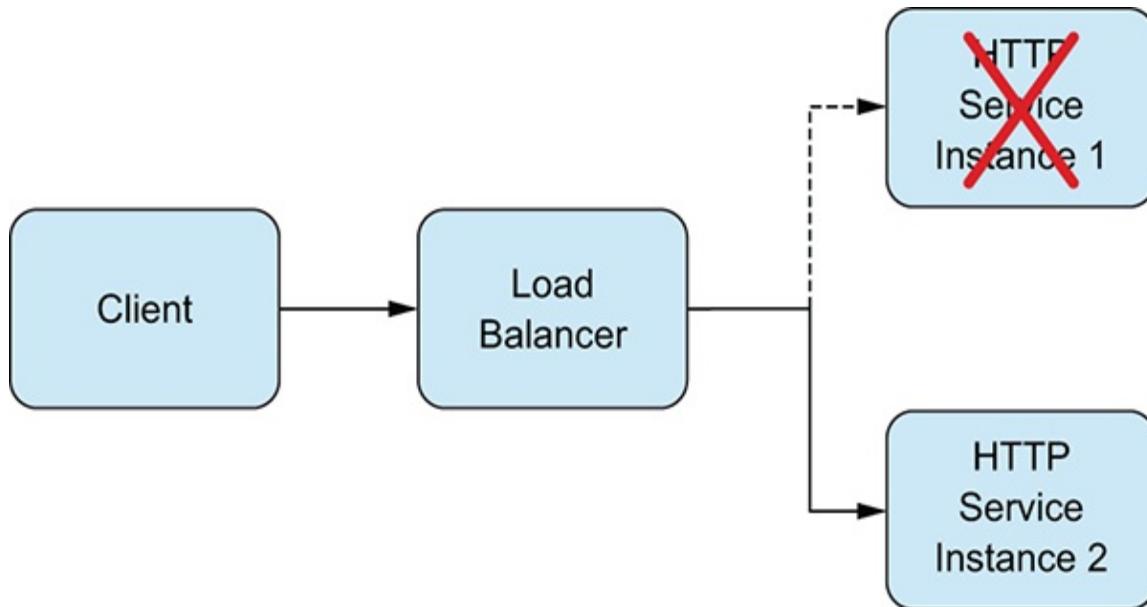


Figure 17.2 Instance 1 of the HTTP service is down, and the load balancer redirects traffic to the available instances (which, in this example, is instance 2).

Using a load balancer requires somehow knowing which instances are currently available so the load balancer can direct traffic to only services that are running. To accommodate this, you often have to implement some kind of heartbeat mechanism—for example, an HTTP endpoint with a *happy page* that returns a positive HTTP status code if the service is ready and alive, and an error code otherwise. The load balancer should be configured to periodically call those HTTP endpoints to obtain the latest status of the instances in the cluster.

Modern infrastructures such as cloud or container platforms often have this mechanism baked into their platform. Chapter 18 covers Kubernetes, which includes readiness and liveness probes as part of its service discovery and load-balancing features.

Stateful HTTP web applications have fallen out of favor in the last decade or so. By *web applications*, we mean Java web applications running on Java EE application servers. The support for clustering came out of the box from the application server. Often all you'd have to do is set up the application servers in a cluster and possibly turn on HTTP session replication.

The usefulness of the HTTP session replication is also questionable to Camel users, as it requires building web applications and storing state in the HTTP session. Camel provides better alternatives for stateful applications, which we talk about in section 17.5.

The next section takes us back in time to talk about an old technique for exchanging data using files or FTP. How do you poll files in a reliable way, using a highly available cluster? It's harder than you may think. How do you ensure that an active node is always polling files? How do you perform failover in case a node dies? These are great questions, and the answers all start with how to cluster Camel routes.

17.2 Clustered Camel routes

When consuming files using the file or FTP component, you can use a couple of strategies for a clustered setup:

- *Active/passive*—Only a single master polls.
- *Active/active*—All nodes poll concurrently.

These two modes will be covered in this section, starting with active/passive mode.

17.2.1 ACTIVE/PASSIVE MODE

In active/passive mode, you have a single master instance polling for files, while all the other instances (slaves) are passive. For this strategy to work, some kind of locking mechanism must be in use to ensure that only the node holding the lock is the master and all other nodes are on standby. [Figure 17.3](#) illustrates active/passive mode.

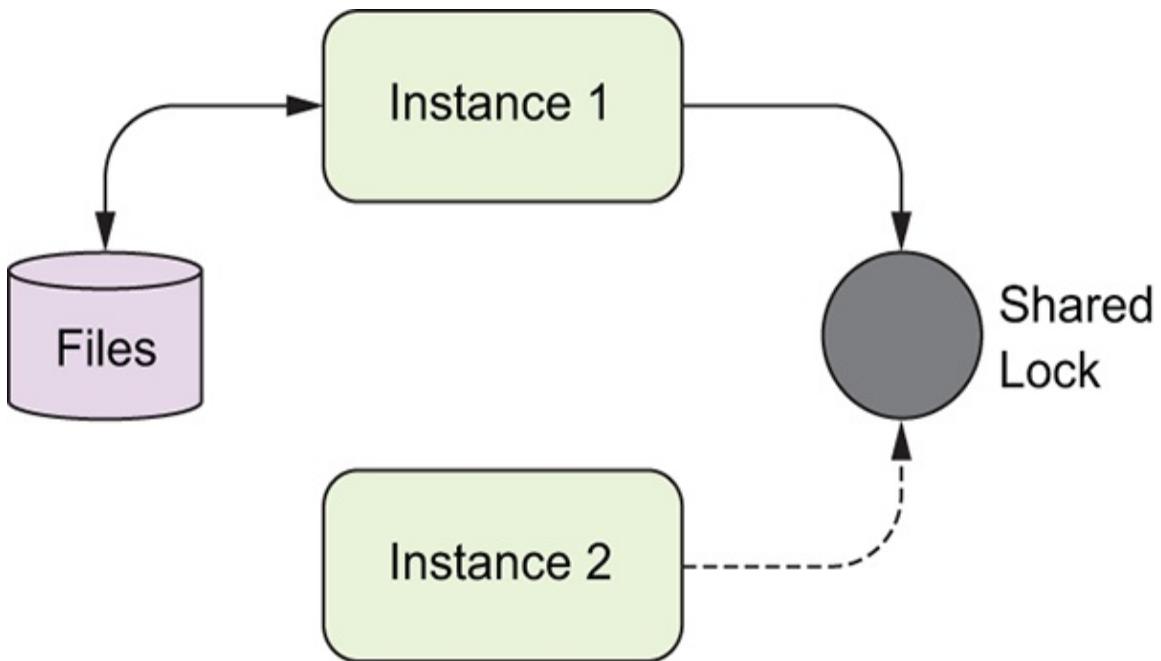


Figure 17.3 Clustering Camel instances using a shared lock in active/passive mode: instance 1 holds the lock and is the master—and therefore is the only instance that's active and polls the files. The other instances are slaves that wait to obtain the lock in order to become the master.

If the master node is shut down or unexpectedly dies, one of the slave instances will obtain the lock and become the new master. Only one node at most is actively polling for files; no concurrent consumers are racing to poll the same files.

When polling files, you often won't need the highest possible performance; for example, if you need to receive the files in batches only once a day, the cluster can handle that with a single instance, which fits perfectly with active/passive mode. But what if you're receiving a continued stream of many files and need to process these files by multiple nodes in the cluster in an active/active mode?

17.2.2 ACTIVE/ACTIVE MODE

In active/active mode, all nodes compete concurrently to pick up and process files. This strategy works by having some kind of clustered repository that keeps track of which files are currently being processed by the nodes in the cluster. [Figure 17.4](#) illustrates active/active mode.

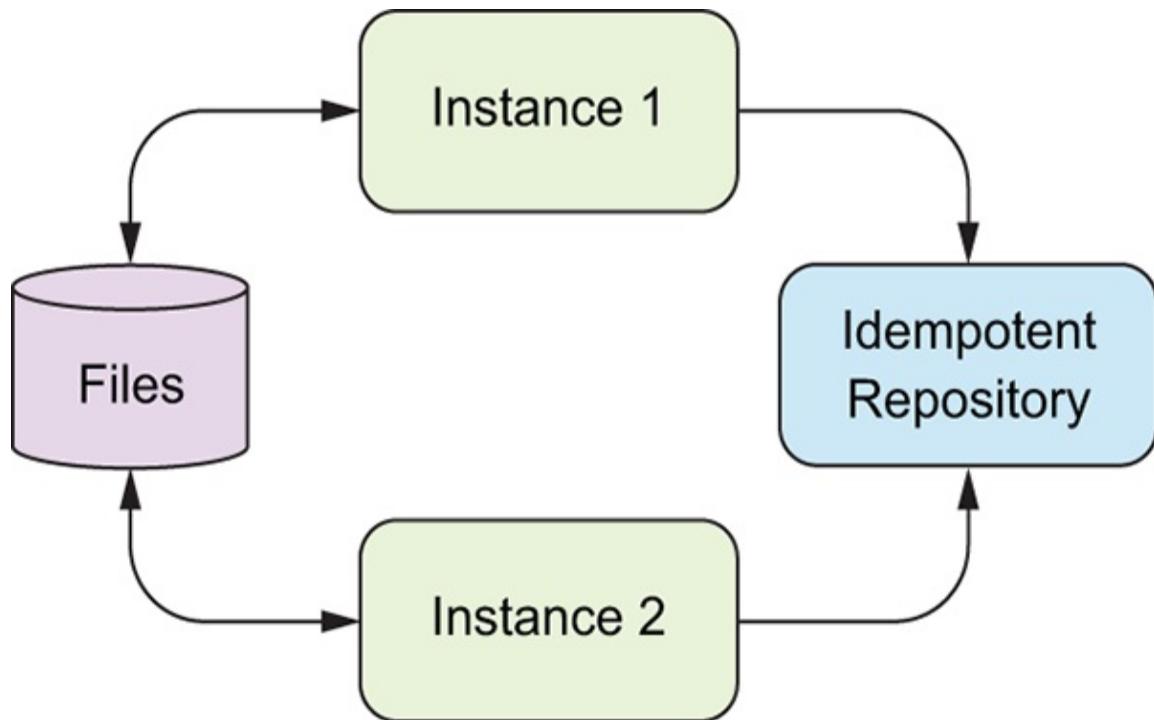


Figure 17.4 Clustering Camel instance using a distributed idempotent repository to track which files are being processed by which nodes. This ensures that a file is being picked up and processed by only one node at most.

When using active/active mode, you can't, for example, use route policy to control the clustering, because it's best suited for active/passive mode. Instead, the file and FTP components allow you to use a distributed idempotent repository, which ensures that only one node is granted access to process the file. Does that mean only one file is processed at any given time? No: all the active nodes compete concurrently to pick up and process files, so each node could pick up a different file, all of which could be processed at the same time. This arrangement yields higher performance, because you can process as many files concurrently as there are nodes.

We covered this use case previously, in chapter 12, section 12.5.3. You can read that section again and review the example if you want to practice working with active/active mode. The remainder of this section focuses on active/passive mode.

How do you set up this active/passive mode with a shared lock? You need a shared lock that supports clustering, and then

you use that with a Camel route policy.

You can use any of the following Camel components that support clustered route policy: camel-consul, camel-etcd, camel-hazelcast, or camel-zookeeper. The following sections use Hazelcast, Consul, and ZooKeeper in the examples.

17.2.3 CLUSTERED ACTIVE/PASSIVE MODE USING HAZELCAST

To represent a cluster with two nodes, we have two almost identical source files in `ServerFoo` and `ServerBar`. The following listing shows the source code for `ServerBar`.

Listing 17.1 Hazelcast route policy in master/slave mode

```
public class ServerBar {  
    private Main main;  
  
    public static void main(String[] args) throws Exception {  
        ServerBar bar = new ServerBar();  
        bar.boot();  
    }  
  
    public void boot() throws Exception {  
        HazelcastInstance hz =  
Hazelcast.newHazelcastInstance(); ①  
    }  
}
```

①

Creates embedded Hazelcast instance

```
HazelcastRoutePolicy routePolicy = new  
HazelcastRoutePolicy(hz); ②
```

②

Creates route policy

```
routePolicy.setLockMapName("myLock"); ③
```

③

Configures route policy

```
routePolicy.setLockKey("myLockKey");      ③  
routePolicy.setLockValue("myLockValue");    ③  
routePolicy.setTryLockTimeout(5,  
TimeUnit.SECONDS);                      ③  
  
main = new Main();  
main.bind("myPolicy", routePolicy);        ④
```

4

Registers route policy in Camel registry

```
main.addRouteBuilder(new FileConsumerRoute("Bar",  
100));          ⑤
```

5

Adds route and runs the application

```
main.run();           ⑤  
}  
}
```

This example runs as a standalone Camel application using a `main` method. At first, you create and embed a Hazelcast instance **1** that by default reads its configuration from the root classpath using the name `hazelcast.xml`. The route policy is then created **2** and configured **3**. It's important to configure the route policy with the same lock name in all the nodes so they're using the same shared lock. The remainder of the code uses Camel `Main` to easily configure with the route policy **4** and run as a standalone Camel application **5**.

The Camel route **5** that uses the route policy is shown in the following listing.

[Listing 17.2](#) Camel route using clustered Hazelcast route policy

```
public class FileConsumerRoute extends RouteBuilder {
```

```
public void configure() throws Exception {  
    from("file:target/inbox?delete=true")  
        .routePolicyRef("myPolicy") ①
```

①

Configures route policy on the route

```
.log(name + " - Received file: ${file:name}")  
.delay(delay) ②
```

②

Delays processing the file so we humans have a chance to see what happens

```
.log(name + " - Done file:      ${file:name}")  
.to("file:target/outbox");  
}  
}
```

The Camel route is a file consumer that picks up files from the target/inbox directory (which is the shared directory in this example). The route is configured by the route policy with the value `myPolicy` ①, which corresponds to the name that the route policy was given in [listing 17.1](#). When a file is being processed, it's logged and delayed ②, so the application runs a bit slower, and you can better see what happens.

You can try this example from the chapter17/cluster-file-hazelcast directory by running the following Maven goals for separate shells so they run at the same time (you can also run the goals from your IDE, because it's a standard Java main application—the IDE typically has a right-click menu to run Java applications):

```
mvn compile exec:java -Pfoo  
mvn compile exec:java -Pbar
```

When you start the second application, Hazelcast should log the cluster state, as shown here:

```
Members [2] {  
    Member [localhost]:5701  
    Member [localhost]:5702 this  
}
```

Here you can see two members in the cluster that are linked together with TCP via ports 5701 and 5702.

If you copy a bunch of files to the target/inbox directory, only the master node will pick up and process the files. While the files are being processed, you can try to shut down or kill the master, which should trigger a failover. The slave node then becomes the master and starts processing the files.

The example can also use Infinispan or Consul instead of Hazelcast. Infinispan and Hazelcast are similar in usage. You can find an example of using Infinispan in the chapter17/cluster-file-infinispan directory, which has further instructions in the readme file for downloading and installing Infinispan (which must be done prior to running the example). The next section covers using Consul.

17.2.4 CLUSTERED ACTIVE/PASSIVE MODE USING CONSUL

HashiCorp Consul is a distributed service registry that also can be used as a clustered lock. You can use Consul in the previous example by making only a few changes. All you have to do is to add camel-consul as a dependency to the project and then use ConsulRoutePolicy, as shown here:

```
ConsulRoutePolicy routePolicy = new ConsulRoutePolicy();  
routePolicy.setServiceName("myLock");  
routePolicy.setTtl(5);
```

You can find this example with the source code in the chapter17/cluster-file-consul directory. To try the example, you need to run Consul, which can be done using Docker:

```
docker run -it --rm -p 8500:8500 consul
```

Consul is then accessible using HTTP on port 8500, which is

what the camel-consul client is using. The Consul web console is also available at <http://localhost:8500/ui>, and you can view that while running the example. Then you can run the two Camel applications by starting two shells and running the following Maven goals:

```
mvn compile exec:java -Pfoo  
mvn compile exec:java -Pbar
```

By copying files to the target/inbox directory, you should see that only one of the Camel applications will pick up and process the files. You can then try to shut down or kill one of the applications and see that Consul fails over to a new master node. From the web console, you should be able to see that Consul also provides details about the distributed lock under the services with the name `myLock`.

TIP Camel registers route policies for Hazelcast, Infinispan, and Consul in JMX, which allows you to see at runtime which node is the master and which are the slaves.

Yet another Camel component supports clustering. The zookeeper-master component uses ZooKeeper to ensure that only a single consumer in a cluster consumes from a given endpoint with automatic failover if that JVM dies.

17.2.5 CLUSTERED ACTIVE/PASSIVE MODE USING ZOOKEEPER

In principle the zookeeper-master component works the same way as Hazelcast and Consul. The ZooKeeper cluster is a distributed and highly available registry that's also capable of orchestrating nodes to conduct in master/slave mode. When your Camel applications are starting, they connect to the ZooKeeper cluster. ZooKeeper then elects one of them as the master, and all other nodes will be slaves on standby. If the master node dies or is shut down, one of the remaining slave

nodes is elected as the new master.

The camel-zookeeper-master JAR provides this functionality both as a Camel component and route policy.

The source code contains two examples. The first example is located in the chapter17/cluster-zookeeper-master directory.

In the cluster, we have yet again two almost identical nodes in the source code as ServerFoo and ServerBar. The following listing shows the source code for the Camel route that is used by both ServerFoo and ServerBar.

[Listing 17.3](#) Camel route using master component for master/slave mode

```
String url = "file:target/inbox?delete=true"; 1
```

1

Consumes files from a shared directory

```
from("zookeeper-master:myGroup:" + url) 2
```

2

[zookeeper-master component to use master/slave mode](#)

```
.log(name + " - Received file: ${file:name}")  
.delay(delay)  
.log(name + " - Done file: ${file:name}")  
.to("file:target/outbox");
```

The Camel route is a simple route that consumes from a shared directory **1**. Notice that the route starts from "zookeeper-master:myGroup:" + url **2**. The route is consuming from the zookeeper-master component in the cluster group with the name myGroup. This lets the zookeeper-master component be in control of the lifecycle of the intended consumer, which is the file consumer. Only if the zookeeper-master component becomes the master will it start up the file consumer.

Before you can try this example, you need to configure the zookeeper-master component, which is done as shown here:

```
MasterComponent master = new MasterComponent();
master.setZooKeeperUrl("localhost:2181"); ①
```

①

Configures URL to ZooKeeper cluster(s)

```
main.bind("zookeeper-master", master); ①
```

②

Registers component in Camel registry

If you're using XML DSL, you configure the component using `<bean>` style, as shown here:

```
<bean id="zookeeper-master"
      class="org.apache.camel.component.zookeeper.master.MasterComponent">
    <property name="zooKeeperUrl" value="localhost:2181"/>
</bean>
```

As you run this example locally, the URL is set to `localhost:2181`. In a real production use case, you'd configure the URL to your ZooKeeper master nodes. You can separate multiple hostnames with commas:

```
keeper1:2181,keeper2:2181,keeper3:2181
```

TRYING THE EXAMPLE

You can try this example from the source code in `chapter17/cluster-zookeeper-master`. First you need to start ZooKeeper, which can easily be run using Docker:

```
docker run -it --rm -p 2181:2181 -d zookeeper
```

Then you can start the Foo and Bar server from two shell commands:

```
mvn compile exec:java -Pfoo  
mvn compile exec:java -Pbar
```

Copy a bunch of files to the target/inbox directory, where only the master node should pick up and process the files. The node that becomes the master will log accordingly:

```
INFO Elected as master. Consumer started:  
file://target/inbox?delete=true
```

While the files are being processed, you can try to shut down or kill the master (i.e. Foo, or Bar servers), which should trigger a failover. The slave node then becomes the master and starts processing the files.

NOTE The source code also contains an example using a route policy instead of a component. You can find this example in the chapter17/cluster-zookeeper-master-routepolicy directory.

What's so special about this zookeeper-master component? It works not only with files but with any Camel consumer endpoint. Because it works in master/slave mode only, the number of use cases is limited. You can use it when you must have at most one active consumer running in your cluster; that's the use case it solves.

SUMMARY OF ACTIVE/PASSIVE VS. ACTIVE/ACTIVE MODE

We've now covered two modes of clustering routes that consume files. The active/passive mode is the easiest and often best choice to process files if a single node can keep up with the volume of files. If you must process a lot of files in a streaming fashion, you may have to use active/active mode to concurrently process the files by all nodes in the cluster. This adds more complexity, because now Camel must use a distributed idempotent repository to orchestrate which files are processed by which nodes.

Camel can also be used with clustering with other protocols such as messaging. The following two sections cover two popular messaging platforms: JMS and Apache Kafka.

17.3 Clustered JMS

JMS clustering with Camel is one of the most difficult scenarios. Many forces are at play, and you have many facets to consider and customize to tailor Camel and the JMS broker and network topology to the business needs.

Most of the clustering functionality with JMS doesn't come from Apache Camel, but from the messaging broker in both its client and server software. The Camel JMS component uses the JMS API and has no concept of clustering in regards to high availability. This section uses Apache ActiveMQ as the message broker, but if you're using a different broker, what you read here is still likely relevant because most message brokers offer similar functionality; certainly all of them support clustering and claim to be highly available.

TIP You can find valuable information about Apache ActiveMQ in *ActiveMQ in Action* by Bruce Snyder et al. (Manning, 2011).

17.3.1 CLIENT-SIDE CLUSTERING WITH JMS AND ACTIVEMQ

When using ActiveMQ, clustering starts from the client point of view, where you can easily turn on support for clustering. This is done using the failover protocol in the URL connection to the broker when configuring the connection factory to ActiveMQ. The following piece of XML illustrates this:

```
<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
```

```
<property name="brokerURL"  
value="failover(tcp://localhost:61616,tcp://localhost:51515  
"/>  
</bean>
```

In this code, the connection factory has been configured using the failover protocol to connect two ActiveMQ brokers running on `tcp://localhost:61616` and `tcp://localhost:51515`. In a real production system, those broker URLs would refer to separate hostnames where the brokers would be running, but because the code is from the book's source code, they use `localhost` to allow you to run this from your computer.

You can find this example in the `chapter17/cluster-jms-client` directory. Before you can run the example, you need to download and run two instances of the ActiveMQ broker on the computer. The example has step-by-step instructions in the accompanying `readme` file.

Upon running this example, you should notice that the client can fail over between the brokers if you shut one of them down. We've pasted output from the client's log showing what happens when we shut down the master ActiveMQ broker (running on `localhost:61616`). The client logs a series of `WARN` messages stating the transport error, followed by an `INFO` log message stating that the client was successfully able to fail over to the new master broker that runs on `localhost:51515`:

```
INFO Successfully connected to tcp://localhost:61616  
INFO Sent message: Time is now Sun Apr 20 21:26:56 CET  
2017  
WARN Transport (tcp://localhost:61616) failed,  
attempting to automatically reconnect  
...  
...  
INFO Successfully reconnected to tcp://localhost:51515  
INFO Sent message: Time is now Sun Apr 20 21:27:02 CET  
2017
```

This is the easiest setup with ActiveMQ, but it does have its restrictions. The ActiveMQ brokers must be using a shared

filesystem that supports file locks. The filesystem must be highly available as well, and therefore you must use a SAN or a hardware appliance providing this kind of functionality.

Highly available messaging is complex

Now you get to the point where you have many facets to consider regarding how to set up a highly available messaging system that your Camel applications can use in the cluster. This topic is complex, and having a full overview takes years for even experts in the field. We advise you to spend diligent time reading the documentation from the broker vendor and other sources. A good blog entry about clustering JMS message brokers is from Josh Reagan:
http://joshdreasan.github.io/2016/07/28/ha_deployments_with_fuse.

Depending on what message broker you use, you should consult its documentation to determine the specific cluster capabilities it provides. For example, Apache ActiveMQ has many features beyond JMS that make it flexible, such as virtual topics. Using virtual topics, you can better cluster, fail over, and scale out than with regular JMS topics.

In recent years, Apache Kafka has become popular. Camel also works great with Kafka, and together, they work with clustering as well.

17.4 Clustered Kafka

Apache Kafka describes itself as a distributed streaming platform that's used for building real-time data pipelines and streaming applications. This section shows you how to stream data from and to Kafka using the camel-kafka component. Kafka is

distributed by nature, and you can fairly easily cluster Camel to reliably consume from Kafka with automatic failover.

This section uses a simple example to demonstrate how to cluster Camel with Kafka. You can run this example locally on your computer. The example, which is in the chapter17/cluster-kafka directory, has four players, as depicted in figure 17.5.

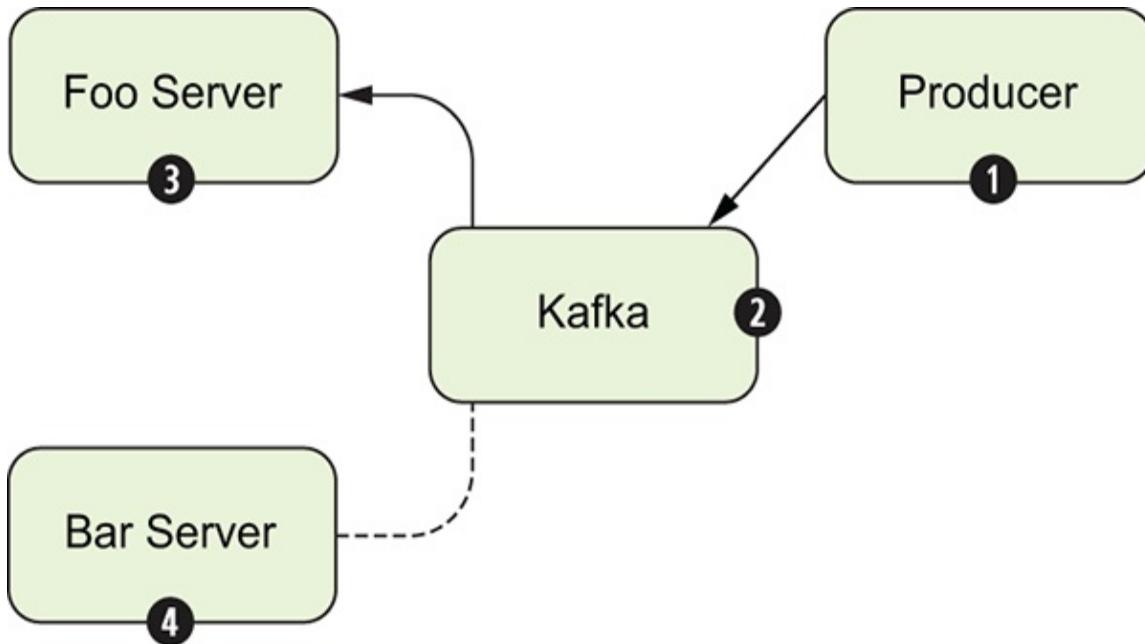


Figure 17.5 The producer ① continuously streams data to the Kafka broker ②. Two Camel servers, Foo ③ and Bar ④, run in clustering mode to consume the events in master/slave mode.

The producer is implemented using a little Camel route that's triggered by a timer that generates a random word that's sent to a Kafka topic.

To use the camel-kafka component, you need to set up the URL to the Kafka broker:

```
KafkaComponent kafka = new KafkaComponent();  
kafka.setBrokers("localhost:9092"); ①
```

①

The URL to the Kafka broker(s)

```
getContext().addComponent("kafka", kafka); ②
```

2

Adds component to CamelContext

In this example, you run Kafka locally and therefore specify the URL as `localhost:9092` ①. Because the example uses Java DSL, you need to add the component to `CamelContext` ②. If you're using XML DSL, you can configure the component as follows:

```
<bean id="kafka"
  class="org.apache.camel.component.kafka.KafkaComponent">
  <property name="brokers" value="localhost:9092"/>
</bean>
```

TIP In a production scenario, you'd set up Kafka in a clustered setup, and therefore the URL to the brokers can contain multiple hostnames separated by commas—for example:
`kafka.setBrokers("kafka1:9092,kafka2:9092,kafka3:9092");`

The Camel route used by the producer is only four lines of code:

```
from("timer:time?period=100")
  .bean(new WordBean())
  .to("kafka:words") ①
```

1

Sends message to Kafka topic: words

```
.to("log:words?groupInterval=1000");
```

When Camel sends a message to Kafka, you need to specify which topic to send the message to. In the example, we send a message to a topic named `words` ①.

About Kafka message key

A Kafka message may contain a key that typically has a semantic meaning, such as a customer ID or item number. The key doesn't have to be unique. It could be specified in the endpoint URI, such as `kafka:words?key=mykey`. You can also specify the key as a header, which allows dynamic values. For example, you can use a header with the customer ID `setHeader("kafka.KEY", header("customerId"))`.

Kafka doesn't require having active consumers running, so you can start the example and let the producer send messages to Kafka. To run Kafka, download Kafka from the Apache Kafka website and run it according to its instructions. You can find details in the readme file with the source code of this example.

In [figure 17.5](#), the two consumers are the Foo **③** and Bar **④** servers that you also implement using Camel, as shown in the following listing.

[Listing 17.4](#) Consuming from Kafka using Camel with the XML DSL

```
<bean id="kafka"
      class="org.apache.camel.component.kafka.KafkaComponent">
    <property name="brokers" value="localhost:9092"/> 1
```

1

Configures URL to the Kafka broker(s)

```
</bean>

<camelContext id="foo"
      xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="kafka:words?groupId=mygroup"/> 2
```

2

Consumes from the Kafka topic: words

```
<log message="Foo got word ${body}" />
</route>
</camelContext>
```

The Kafka component must be configured with the URL to the Kafka broker, which is done using the <bean>-style configuration
1. The Camel route is consuming from the Kafka topic named words **2**, and each message is then logged. The option groupId=mygroup is used to group consumers into the same group across the cluster. The group name is global across the cluster, so be sure to use the correct name.

You're now ready to run the example from the chapter17/cluster-kafka directory and start the producer and the first of the two consumers, which is the Foo server. This is done using separate command shells and executing the following Maven goals:

```
mvn compile exec:java -P producer
```

That starts the producer, which starts sending messages to Kafka. The consumer is started as follows:

```
mvn compile exec:java -P foo
```

When the consumer is up and running, you should see the receive messages being logged:

```
INFO  Foo got word #1-Dude
INFO  Foo got word #2-Rocks
INFO  Foo got word #3-Dude
INFO  Foo got word #4-Bad
INFO  Foo got word #5-Bad
INFO  Foo got word #6-Fabric8
```

You haven't yet clustered Camel with Kafka, because you have only one Camel consumer running. Let's see what happens when you start the second consumer:

```
mvn compile exec:java -P bar
```

When the second consumer starts, it joins the Kafka broker as a consumer on the topic named words with the group ID mygroup.

Because a consumer is already running on that same topic with the same group ID, two things happen:

- *Kafka repartitions the connected consumers*—Because you have only two consumers and one partition, only one consumer can receive messages while the other is on standby. You have a master/slave scenario.
- *Whenever Kafka repartitions, it can decide to assign partitions to new consumers*—Either the existing or the new consumer will be assigned the partition. The one who has been assigned will be the only active consumer that receives the messages.

The following snippet shows such a situation; Kafka has reassigned the partition to the second consumer (Bar):

```
INFO Successfully joined group mygroup with generation 22
INFO Setting newly assigned partitions [words-0] for group
mygroup
INFO Bar got word #269-Beer
INFO Bar got word #270-Cool
```

Kafka will also repartition when a consumer is stopped. For example, if you try to stop the active consumer, you should see the other consumer take over. You've achieved clustering Camel with Kafka in a way that works with automatic failover.

Figure 17.6 illustrates this failover action.

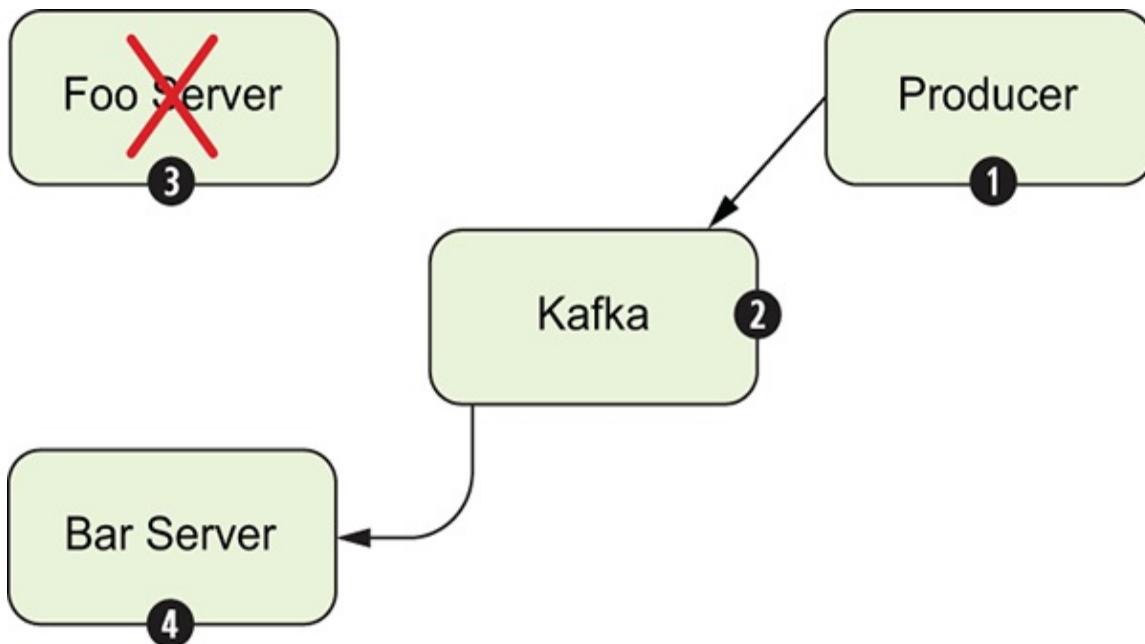


Figure 17.6 Kafka failover in action: the Foo server ❸ was the previously active consumer that received messages from the Kafka topic. But the server was stopped, and Kafka repartitioned the topic among the remaining consumers. Only the Bar server ❹ is left, which then becomes the new active consumer and starts receiving messages.

Kafka is built using a different architecture than traditional message brokers. With JMS messaging, the broker plays a central role in distributing messages fairly and reliably among consumers, and JMS clients have to worry about only sending and receiving messages. Kafka, on the other hand, is client-centric. Clients take over many of the functionalities of traditional brokers, handling duplicate messages and dealing with other kinds of errors.

Next we'll look at one of these client responsibilities that affects running Kafka with Camel. With Kafka, the client must keep track of which messages it has received, known as the *consumer offset*.

17.4.1 KAFKA CONSUMER OFFSET

With Kafka, you may get duplicate messages, and why is that? Whenever a consumer consumes from a Kafka topic, the consumer is responsible for keeping track of which messages it has received; this record is called an *offset*. The consumer

periodically commits this offset to Kafka so it can resume from this offset in the event of a restart or failover. By default, camel-kafka commits this offset every 5 seconds (autocommit). The offset is also committed when Camel is stopped—for example, when you stop the Camel application gracefully.

TIP From Camel version 2.21 onward you can control consumer offset commits manually as well by using the `KafkaManualCommit` API. See the camel-kafka documentation from the Camel website for more details.

You can see this by starting and stopping the consumers in the example. You should notice that no duplicate messages occur when Camel fails over.

In the following snippet, the Foo server is stopped, and Bar is taking over:

```
INFO  Foo got word #118-Cool
INFO  Foo got word #119-Bad
INFO  Auto commitSync on stop words-Thread 0 from topic
words
```

Here are logs from Bar:

```
INFO  Setting newly assigned partitions [words-0] for group
mygroup
INFO  Bar got word #120-Hawt
INFO  Bar got word #121-Dude
```

Notice that the last message processed by Foo is number 119. Upon stopping Foo, the offset is committed, which is what that last line means. When Bar takes over, the offset is 119, and therefore Bar can continue where Foo left off (at the next message, 120, and so on).

Okay, this is awesome, so we're all good? Frankly, we're not, because this works reliably only when Kafka consumers are shut down gracefully. A hard stop such as a JVM crash isn't a graceful shutdown, and what happens then? Why not have fun and try

this?

17.4.2 CRASHING A JVM WITH A RUNNING KAFKA CONSUMER

What you want to do now is to let both the Foo and Bar servers run at the same time, kill the JVM that has the active Kafka consumer, and then see what happens when the other consumer is failing over.

First you start the producer so the consumers will have messages to consume from Kafka:

```
mvn compile exec:java -P producer
```

Then you start both consumers:

```
mvn compile exec:java -P foo  
mvn compile exec:java -P bar
```

Now you want to kill the running consumer, which can be either the Foo or Bar server. In this example, you'll assume that the Foo server is the active consumer. To kill a JVM, you need its process ID (PID), which you can find using the Java jps tool, as shown here:

```
$ jps -m  
13138 QuorumPeerMain config/zookeeper.properties  
14835 Launcher compile exec:java -P producer  
14837 Launcher compile exec:java -P foo  
14839 Launcher compile exec:java -P bar  
14873 Jps -m  
13358 Kafka config/server.properties
```

As you can see, the process ID for the Foo server is 14837, which we can kill using the following:

```
kill -9 14837          (using OSX or Linux)  
taskkill /PID 14837 /F    (using Windows)
```

When the Foo server is killed, it stops processing, and the last log is shown here:

```
INFO  Foo got word #4640-Bad
```

```
INFO Foo got word #4641-Hawt  
Killed: 9
```

You then wait for the Bar server to fail over and start processing the messages. The failover doesn't happen immediately, as it did previously when you stopped the Foo server gracefully. Instead, the Kafka broker has to realize that the consumer has crashed. Every Kafka consumer that's part of a cluster group will use a heartbeat to tell the Kafka brokers that it's alive. But if a consumer hasn't successfully reported a heartbeat within a 10-second time-out (session time-out), the consumer is considered dead. Therefore, it will take 10 seconds or longer before Kafka repartitions the topic and assigns it to the Bar consumer. This is what happens in the following snippets of the Bar server's logs when it starts processing the messages:

```
INFO Successfully joined group mygroup with generation 4  
INFO Setting newly assigned partitions [] for group  
mygroup  
INFO (Re-)joining group mygroup  
INFO Successfully joined group mygroup with generation 5  
INFO Setting newly assigned partitions [words-0] for group  
mygroup  
INFO Bar got word #4627-Donkey  
INFO Bar got word #4628-Camel  
INFO Bar got word #4629-Donkey  
INFO Bar got word #4630-Dude
```

As you can see, the Bar consumer was assigned the partition of the words topic and then started receiving the messages.

The first message received by the Bar consumer is #4627-Donkey, and the last message sent by the Foo server before it crashed is #4641-Hawt. You'll receive duplicate messages for messages 4627 through 4641. This duplication occurs because the Foo consumer performs a periodical autocommit of the consumer offset every 5 seconds. Therefore, a new consumer that has been assigned a partition because the previous consumer died risks duplicate messages going back for 5 seconds.

Apache Kafka Books

If you need to use Apache Kafka, we recommend studying and reading relevant Kafka material such as tutorials, videos, and books. In particular, if you come from a traditional messaging background, you'll see that Kafka does things differently than those messaging systems.

Here are two books that we recommend:

Kafka: The Definitive Guide by Neha Narkhede (O'Reilly, 2017)

Understanding Message Brokers by Jakub Korab (free ebook by O'Reilly)

That concludes the most commonly used component for clustering. The next section covers specifics about clustered caches with Camel.

17.5 Clustering caches

Writing stateful applications in a distributed and clustered system is more complex because the state must be shared to all the nodes in the cluster. This is what *clustered memory grid* (clustered cache) systems such as Hazelcast, Infinispan, or Redis are designed to solve.

Let's illustrate this with a simple example, shown in [figure 17.7](#).

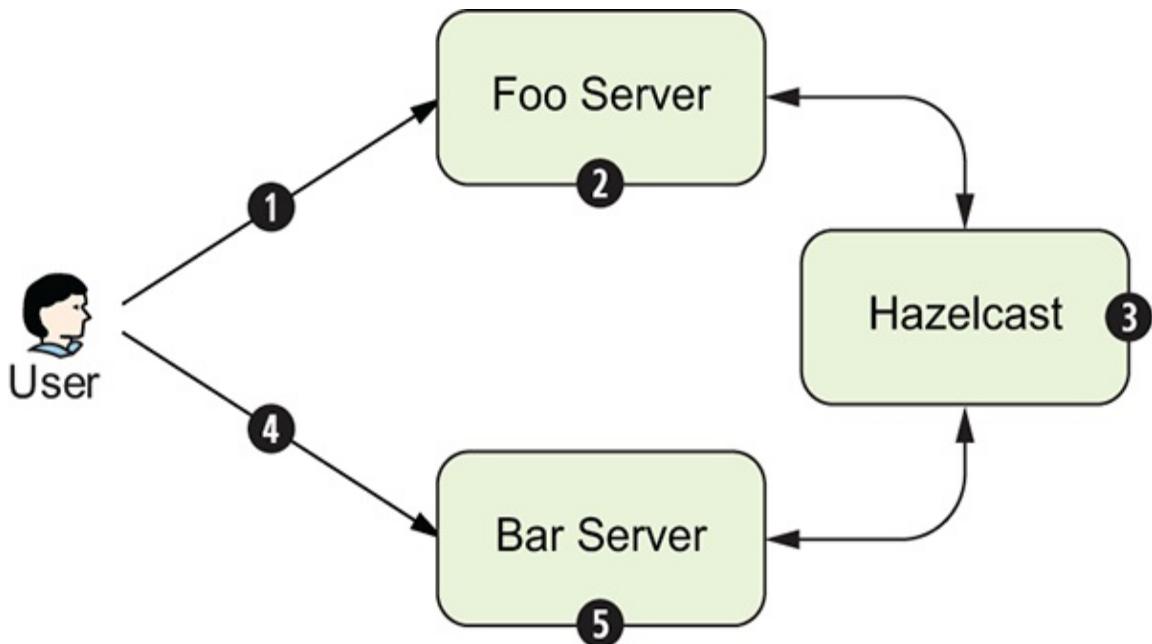


Figure 17.7 A user accesses a distributed Camel application that has been clustered and runs on two nodes (2, 5). The application stores state about user activity, which is made available at all times using the clustered cache provided, such as Hazelcast (3).

The same Camel application is deployed and running on both servers 2 5 in a clustered setup. This is done because the application must be made highly available in case one of the servers becomes unavailable. In this example, either of the two servers can be accessed by the client 1 4, which would require any data that must be shared between the nodes to be made available on both nodes. This is where the clustered cache from Hazelcast 3 enters the scene and provides such functionality.

17.5.1 CLUSTERED CACHE USING HAZELCAST

As a Camel developer, you shouldn't worry too much about how Hazelcast clustering works. Just use camel-hazelcast like any other Camel component.

In this example, the client calls the application using HTTP. The stateful data is a counter value that's incremented on each call and is returned as a response to the client. The following listing shows the source code of the Camel route implementing this.

Listing 17.5 Stateful Camel application using Hazelcast to distribute state in the Camel cluster

```
fromF("jetty:http://localhost:" + port) ①
```

1

Starts the route using Jetty as HTTP server

```
.setHeader(HazelcastConstants.OBJECT_ID,  
constant("myCounter")) ②
```

2

Gets the shared data from Hazelcast

```
.to("hazelcast:map:myCache?  
hazelcastInstance=#hz&defaultOperation=get") ②
```

```
.process(new Processor() { ③
```

3

Updates the shared data

```
    public void process(Exchange exchange) throws Exception  
{  
        Integer counter =  
exchange.getIn().getBody(Integer.class);  
        if (counter == null) {  
            counter = 0;  
        }  
        counter++;  
        exchange.getIn().setBody(counter);  
    }  
})  
.setHeader(HazelcastConstants.OBJECT_ID,  
constant("myCounter")) ④
```

4

Stores the shared data back to Hazelcast

```
.to("hazelcast:map:myCache?  
hazelcastInstance=#hz&defaultOperation=put") ④  
  
.log(name + ": counter is now ${body}")  
.setHeader(Exchange.CONTENT_TYPE,  
constant("text/plain")) ⑤
```

⑤

Prepares the HTTP response message

```
.transform().simple("Counter is now ${body}\n"); ⑤
```

The Camel route starts with the Jetty endpoint that exposes an HTTP service running on `localhost` ①. The shared state is then accessed from the Hazelcast cluster using the `camel-hazelcast` component. A header must be configured with the name of the state (object ID) before calling the endpoint ②. We've highlighted in bold the pieces that make up what Camel is asking of Hazelcast:

```
.setHeader(HazelcastConstants.OBJECT_ID,  
constant("myCounter"))  
.to("hazelcast:map:myCache?  
hazelcastInstance=#hz&defaultOperation=get")
```

As you can see, you get from a Hazelcast map named `myCache` the value of the key `myCounter`.

The Camel route uses a processor ③ to update the counter, which is then stored back into Hazelcast ④. The last piece of the route is to prepare the response to return to the user ⑤.

You can find this example in the `chapter17/cluster-cache-hazelcast` directory. To try this example, you need to run two JVMs, each with the Foo or Bar server, which can be started from Maven as follows:

```
mvn compile exec:java -Pfoo  
mvn compile exec:java -Pbar
```

From a web browser, you can call either the Foo or Bar server

with the following URLs: <http://localhost:8080> and <http://localhost:9090>. You should then be able to receive a response with an increasing counter that's distributed. If you mix the calls between the two JVMs, the counter is correctly increased by one each time. You can try to stop one of the servers and bring it back up, which should let it reconnect to the Hazelcast cluster and work again.

The perceptive reader may have spotted a problem with the example source code in [listing 17.5](#). At first you get the current counter value ②, which is then updated by Camel ③ and stored back again ④. What happens if both servers concurrently read the counter at the same time? For example, if both servers read the counter as 42, they'd both update and set the counter to 43. But the counter was supposed to be 44. What happened is called the *lost update problem*.

Storing stateful data in a map is good for many uses cases, such as tracking user activity or items in a shopping cart. But this example was purposefully designed to make you aware of the problems with counters. Instead of using a map, you should use *clustered* counters.

CLUSTERED COUNTERS

Some distributed memory grid systems have support for atomic counters. For example, Hazelcast has this support, and it's on the roadmap for Infinispan 9.x. Using a clustered counter with camel-hazelcast is easy. The source code from [listing 17.5](#) can be shortened to just one call to Hazelcast:

```
fromF("jetty:http://localhost:" + port)
    .setHeader(HazelcastConstants.OBJECT_ID,
constant("myCounter"))
    .to("hazelcast:atomicvalue:Cache?
        hazelcastInstance=#hz&defaultOperation=increment")
    .log(name + ": counter is now ${body}")
    .setHeader(Exchange.CONTENT_TYPE, constant("text/plain"))
    .transform().simple("Atomic Counter is now ${body}\n");
```

Instead of using Hazelcast `map`, you use `atomicvalue`. The counter

is updated using the `increment` operation, which works safely in the cluster.

You can try this variation of the example by running the Foo and Bar servers with the following Maven goals:

```
mvn compile exec:java -Pfoo-atomic  
mvn compile exec:java -Pbar-atomic
```

The examples using Hazelcast in this chapter have all been using embedded mode. This means you've colocated Hazelcast with your Camel application. In the real world, you'd use a separate Hazelcast cluster that you can manage and run independently from your Camel applications. The next section covers an example of using a separate cluster with Camel, except instead of using Hazelcast, you'll use Infinispan, and use JCache as the abstraction API between Camel and the cache.

17.5.2 CLUSTERED CACHE USING JCACHE AND INFINISPAN

JCache is a standard that provides a common set of caching APIs that allow clients to access different caching providers using the same API standard. This should allow you to use the `camel-jcache` component and then plug in different Cache providers such as Hazelcast or Infinispan without having to change your Camel route. This example uses a clustered Infinispan setup. You can do this on your own by downloading Infinispan Server from <http://infinispan.org>. Further instructions for setting up Infinispan are provided in the source code in the `chapter17/cluster-jcache` directory.

When using JCache, you must specify which `JCachingProvider` you're using and its configuration. To use Infinispan, you need to use the `org.infinispan.jcache.remote.JCachingProvider` provider, which you find in the `infinispan-jcache-remote` JAR file.

The Camel application is represented with almost identical source code in the `ServerFoo` and `ServerBar` classes. The

following listing shows the source code for ServerBar.

Listing 17.6 Setting up JCache component to use Infinispan remote server

```
public class ServerBar {  
    private Main main;  
  
    public static void main(String[] args) throws Exception {  
        ServerBar bar = new ServerBar();  
        bar.boot();  
    }  
  
    public void boot() throws Exception {  
        main = new Main();  
  
        JCacheComponent jcache = new JCacheComponent(); 1  
    }  
}
```

1

Creates Camel JCache component

```
jcache.setCachingProvider(JCachingProvider.class.getName()); 2  
;
```

2

Uses Infinispan provider

```
jcache.setConfigurationUri("hotrod-  
client.properties"); 3
```

3

Specifies the Infinispan configuration file

```
main.bind("jcache", jcache); 4
```

4

Uses jcache as the component name

```
        main.addRouteBuilder(new CounterRoute("Bar", 8889));
        main.run();
    }
}
```

The camel-jcache component requires you to choose a JCache provider, such as Infinispan or Hazelcast. You specify the chosen provider and, optionally, additional configurations as options on the camel-jcache component, as shown in [listing 17.6](#). At first the JCacheComponent is created ①, and then the provider from Infinispan is configured ②. Because you're using a remote Infinispan cluster, you must configure the provider with details such as the URLs to the remote Infinispan cluster and other relevant settings. This example uses the Infinispan Java client, which is named hotrod, and hence the configuration name is hotrod-client.properties ③. To keep the example simple, the configuration file has only one line:

```
infinispan.client.hotrod.server_list=localhost:11222;localhost:11372
```

After the component has been configured, it's added to the Camel register using the name jcache ④.

The Camel route is similar to what you've seen when using Hazelcast, such as in [listing 17.5](#). The following code snippet shows you how to get from the cache:

```
.setHeader(JCacheConstants.KEY, constant("myCounter"))
.to("jcache:myCache?action=get")
```

And the following shows how to update the cache:

```
.setHeader(JCacheConstants.KEY, constant("myCounter"))
.to("jcache:myCache?action=put")
```

If you've successfully set up the Infinispan cluster (remember to create myCache using the Infinispan web console), you should be able to run the two Camel applications with Maven from the chapter17/cluster-jcache directory:

```
mvn compile exec:java -Pfoo
mvn compile exec:java -Pbar
```

The example works similarly to the Hazelcast examples. You call either `http://localhost:8888` or `http://localhost:8889` to access the servers, and the response should include the counter going up by one number for each call.

The last clustered component we want to discuss is the Quartz scheduler, used to schedule Camel routes to run clustered.

17.6 Using clustered scheduling

The Quartz component is used for scheduling Camel routes to run at certain intervals. We covered Quartz in chapter 6, section 6.7. This time, we'll show you how to use Quartz in a clustered setup. You may want to do this when you need running tasks periodically or according to a cron expression (cron was covered in section 6.7.2). Suppose you want to run a task every minute during opening hours of the business (8 a.m. to 6 p.m.), every day. Here's the cron expression for doing this:

```
0 0/1 08-18 ? * *
```

Now suppose you have a cluster of two nodes, each running a Camel application, and you want to run this job on only one node. You must run this task in master/slave mode, and you can implement that using a clustered scheduler, which means using the Quartz component.

17.6.1 CLUSTERED SCHEDULING USING QUARTZ

To use Quartz in clustering, you need to do the following three tasks:

- Set up and prepare a shared database
- Configure Quartz to use the database
- Define Camel route(s) using the Quartz component

An example with the source code is located in the `chapter17/cluster-quartz` directory.

SETTING UP DATABASE FOR QUARTZ

Quartz supports most common databases, and we've chosen to use Postgres. To quickly run Postgres, you can run the following Docker command:

```
docker run -p 5432:5432 -e POSTGRES_USER=quartz -e POSTGRES_PASSWORD=quartz -d postgres
```

This starts Postgres and binds the network listener of Postgres on port 5432. This will allow you to log in to the database using the supplied username and password. The name of the database is the same name as the username, which is quartz. You then need to prepare the database to create the necessary tables that Quartz uses to store its clustered state. If you download the Quartz distribution, you can find the SQL scripts for the supported databases in the docs/dbTable directory. We've made it easy to set up the Postgres tables using Java. You can run the following Maven goal from the chapter17/cluster-quartz2 directory:

```
mvn compile exec:java -P init
```

CONFIGURING QUARTZ TO USE THE DATABASE

Quartz must be configured to use the database, which is done in the quartz.properties file. The most noteworthy configuration is shown here:

```
org.quartz.scheduler.instanceId = AUTO
org.quartz.jobStore.class =
org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.driverDelegateClass = org.quartz.impl.

jdbcjobstore.PostgreSQLDelegate
org.quartz.jobStore.dataSource = quartzDataSource
org.quartz.jobStore.isClustered = true
org.quartz.dataSource.quartzDataSource.driver =
org.postgresql.Driver
org.quartz.dataSource.quartzDataSource.URL =

jdbc:postgresql://localhost:5432/quartz
org.quartz.dataSource.quartzDataSource.user = quartz
```

```
org.quartz.dataSource.quartzDataSource.password = quartz
```

If you use a different database, you must configure the file according to the name of the JDBC driver and the URL.

CAMEL ROUTE USING QUARTZ

Using the Camel Quartz component is easy—all you need to do is to configure the component with the location of the `quartz.properties` file:

```
QuartzComponent quartz = new QuartzComponent();
quartz.setPropertiesFile("quartz.properties");
main.bind("quartz2", quartz);
```

Then the Quartz component is used in the Camel route to trigger according to the desired cron expression:

```
from("quartz2:myGroup/myTrigger?cron=0+0/1+08-18+?+*+*")
    .log(name + " running at ${header.fireTime}");
```

You can run this example by starting two Camel applications known as Foo and Bar using the following Maven goals:

```
mvn compile exec:java -P foo
mvn compile exec:java -P bar
```

Then you should notice that only one node will ever run the task at every minute. Quartz will let only the node that holds the database lock run the task. The lock is then released after the task is complete. At the next scheduled time, both nodes will race to acquire the lock, and whichever grabs the lock runs the task. (The node that runs the task is determined randomly.) If a node crashes, the other node will be there to run the task. You have high availability as long you have at least one running node, and the database must also be running. How to make the Postgres database clustered is beyond the scope of this book. This brings us to the caveat of using Quartz in clustered mode: Quartz must use a database.

The last part of this chapter covers the newest addition (at the time of this writing) to Camel's EIP patterns: the Service Call

EIP.

17.7 Calling clustered services using the Service Call EIP

The Service Call EIP is a new EIP pattern added in Camel 2.18. This pattern is used for calling remote services in a distributed system. The pattern has the following noteworthy features:

- *Location transparency*—Decouples Camel and the physical location of the services using logical names representing the services.
- *URI templating*—Allows you to template the Camel endpoint URI as the physical endpoint to use when calling the service.
- *Service discovery*—Looks up the service from a service registry of some sort to know the physical locations of the services.
- *Service filter*—Allows you to filter unwanted services (for example, blacklisted or unhealthy services).
- *Service chooser*—Allows you to choose the most appropriate service based on factors such as geographical zone, affinity, plans, canary deployments, and SLAs.
- *Load balancer*—A preconfigured Service Discovery, Filter, and Chooser intended for a specific runtime (these three features combined as one).

In a nutshell, the EIP pattern sits between your Camel application and the services running in a distributed system (cluster). The pattern hides all the complexity of keeping track of all the physical locations where the services are running and allows you to call the service by a name.

This is an oversimplification of what happens, so let's dive deep into the innards of Camel and see how this EIP works.

17.7.1 HOW THE SERVICE CALL EIP WORKS

Figure 17.8 depicts the algorithm in use by the Service Call EIP when a service is to be called.

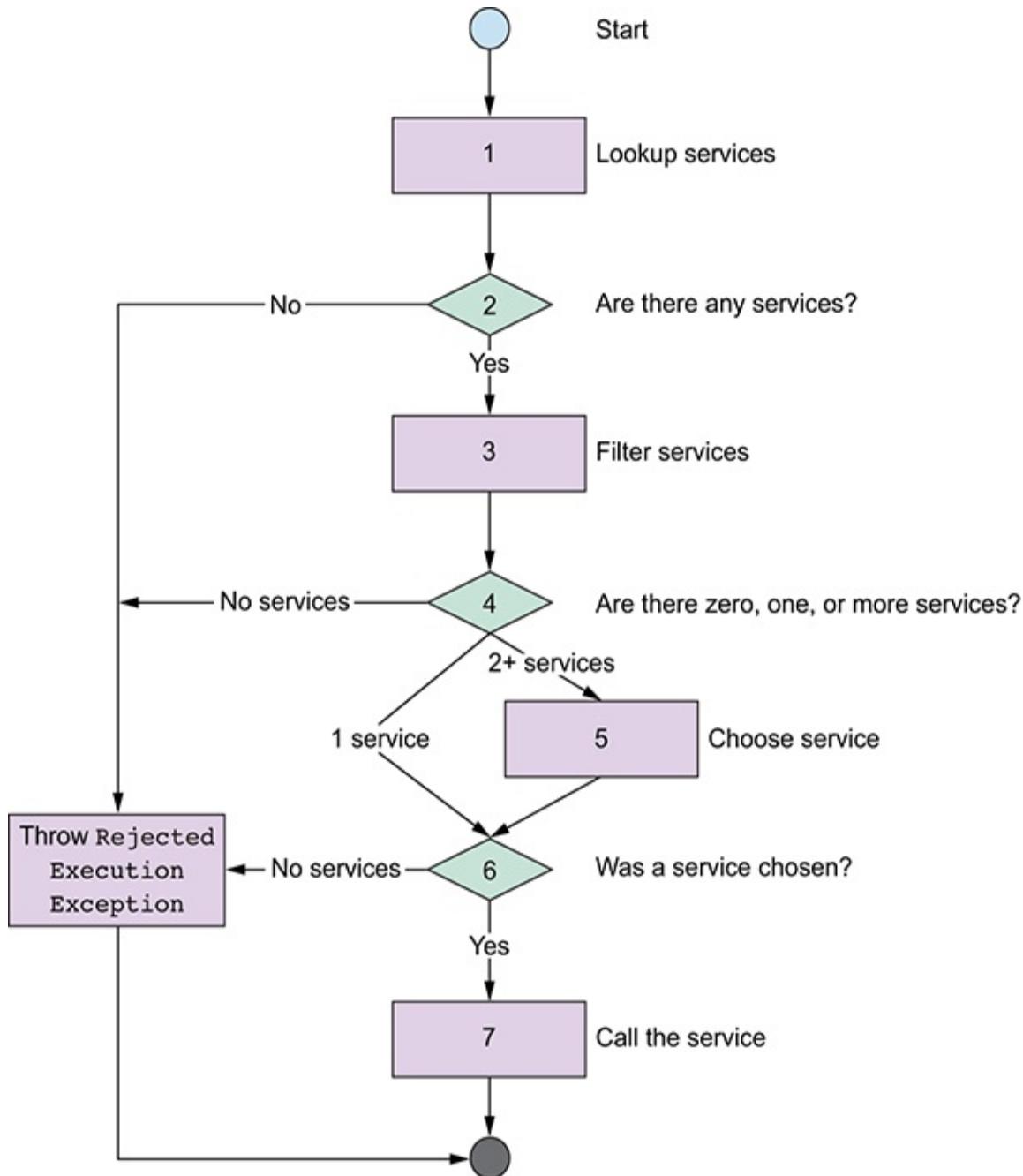


Figure 17.8 Flow chart of how the Service Call EIP works when calling a service

Here's the algorithm for the way the Service Call EIP works:

1. The logical name of the service to call is used to look up from an external registry the physical locations where the service is

hosted.

2. The lookup returns a list of physical locations of the service. If the list is empty, `RejectedExecutionException` is thrown and the call is ended. If the list contains numerous services, go to step 3.
3. The services are filtered to remove unwanted services. For example, some services may be blacklisted. A typical filter would be to include only services considered healthy. This requires using health checks, which we talk about in chapter 18.
4. After the filtering, are there zero, one, or more services? If there are no services, `RejectedExecutionException` is thrown, and the call is ended. If there's only one service, go to step 7.
5. When two or more services exist after the filtering, you need to choose one. For example, if the client has called the service before, you could try to choose the same physical service again (affinity). Another criteria could be to choose the physical service located in the same data center, or in the same geographical region. Or you can use old-fashioned round-robin or randomly choose a service.
6. Was a service chosen? If not, `RejectedExecutionException` is thrown, and the call is ended. If a service was chosen, go to step 7.
7. A service has been chosen and is about to be called. When Camel calls the service, it maps the physical location of the service to a Camel endpoint URI using URI templating (it's like Camel's `<toD uri="..."/>` to call a dynamic endpoint). This allows extreme flexibility, and to use any of the Camel components as a producer calling the service.

Now it's time to take a look at how this algorithm applies in practice.

17.7.2 SERVICE CALL USING STATIC SERVICE REGISTRY

Let's start easy and use an example that has a client that calls a clustered service. The cluster is static and hosts the service on

two running nodes (Foo server and Bar server). Figure 17.9 depicts this scenario.

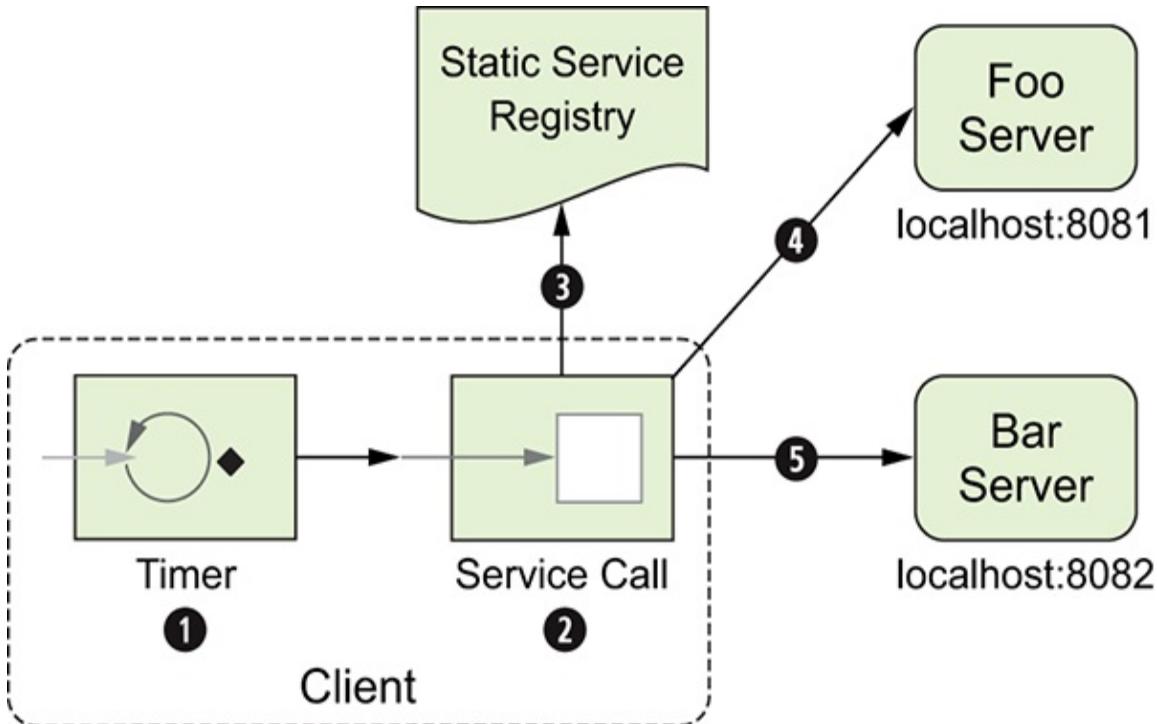


Figure 17.9 A timer ① triggers the Service Call EIP ② to call a clustered service. The physical locations of the service are looked up in the service registry ③. The service is then called in a round-robin fashion by calling either Foo server ④ or Bar server ⑤.

The client is developed as a Camel route that starts from a timer ①, which then calls the service ②. The source code for such a route in Java and XML DSL is as follows:

```
from("timer:trigger")
    .serviceCall("hello-service")
```

In XML DSL:

```
<route>
    <from uri="timer:trigger"/>
    <serviceCall name="hello-service"/>
</route>
```

There's more to this: you need to configure the Service Call EIP with a strategy for discovering and looking up the services, as well as for choosing one service among the available services (for

example, according to the algorithm laid out in [figure 17.8](#)).

The example uses a static list of services **③**, which is configured using the syntax

serviceName@hostname:port, serviceName@hostname2:port2 (you separate multiple host configurations with commas). The name of the service is `hello-service`, which gives the following server configuration:

```
hello-service@localhost:8081, hello-service@localhost:8082
```

The static list of servers contains two physical locations **(4)** **(5)**, but the Service Call EIP needs to call only one of them. Therefore, one is chosen, and it's chosen in round-robin fashion by default. Having chosen one server, the EIP constructs the Camel endpoint URI, using URI templating, to use when calling the service. We'll go over this in more detail in a moment. But first, the following listing shows you the source code for this example.

Listing 17.7 Using Service Call EIP to call a clustered service (static) in Java DSL

```
public class MyStaticRouteEmbedded extends RouteBuilder {  
  
    public void configure() throws Exception {  
        from("timer:trigger?period=2000")  
            .serviceCall() ①
```

1

Service Call EIP

```
        .component("http4") ②
```

2

Using http4 component

```
        .name("hello-service/camel/hello") ③
```

③

Configures name of service

```
.staticServiceDiscovery() ④
```

④

Static location of services

```
.servers("hello-service@localhost:8081," ④  
        + "hello-service@localhost:8082")  
.end() ⑤
```

⑤

Ends configuration of static list

```
.end() ⑥
```

⑥

Ends Service Call EIP

```
.log("Response ${body}");  
}  
}
```

The route starts from a timer that triggers every other second. Then the Service Call EIP block begins ① and ends ⑥. The component ② is used to specify which Camel component to use when calling the service. In this example, we use HTTP as a protocol and can therefore use any of the many HTTP components from Camel.

TIP The Service Call EIP uses http4 as the default component, so you could leave out this configuration in this example.

The name ③ of the service must be configured. The configured name is specified as hello-service/camel/hello. The name

supports URI templating in the form of name/context-path? parameters. This means the name of the service is `hello-service` and with an associated `/camel/hello` as context-path.

To keep the example simple, we specify the physical location of the two servers in a static list ④. Notice that we use `end()`s to end both the static list ⑤ and the Service Call EIP ⑥.

You can also use Service Call in XML DSL, as shown in the next listing.

Listing 17.8 Using Service Call EIP to call a clustered service (static) in XML DSL

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

  <route>
    <from uri="timer:trigger?period=2000"/>
    <serviceCall name="hello-service/camel/hello"
      component="http4"> ①
```

①

Service Call EIP

```
    <staticServiceDiscovery>
      <servers>
        hello-service@localhost:8081, hello-
        service@localhost:8082 ②
```

②

Configures static list of servers

```
    </servers>
  </staticServiceDiscovery>
</serviceCall>
<log message="Response ${body}" />
</route>

</camelContext>
```

In XML DSL, you specify the service name and component as

attributes on the <serviceCall> element ❶. You specify the static list of servers ❷ in similar way to Java DSL, where it's embedded inside the Service Call EIP.

RUNNING THE EXAMPLE

You can try this example by running the following three Java applications from separate command shells. Running the Foo server:

```
cd chapter17/cluster-servicecall/foo-server  
mvn spring-boot:run
```

Running the Bar server:

```
cd chapter17/cluster-servicecall/bar-server  
mvn spring-boot:run
```

Running the client:

```
cd chapter17/cluster-servicecall/client-static  
mvn camel:run -P embedded
```

You can also try the XML DSL version of the client by running this:

```
cd chapter17/cluster-servicecall/client-static-xml  
mvn camel:run -P embedded
```

When running the example, the client will output responses from the two servers in round-robin mode:

```
INFO Response Hello from Foo server on port 8081  
INFO Response Hello from Bar server on port 8082  
INFO Response Hello from Foo server on port 8081  
INFO Response Hello from Bar server on port 8082
```

This looks good, and it's hardly a surprise that Camel is able to call two local servers. But let's see what happens if one of these servers is down.

17.7.3 SERVICE CALL WITH FAILOVER

In chapter 7, we talked about how you should design for failures

with distributed systems. You can argue we have a mini distributed system in the example in [figure 17.9](#), where the hello service is distributed among the two servers (Foo and Bar). When the client attempts to call the service using the Service Call EIP, a failure can happen. Let's see what happens if, for example, the Foo server is unavailable.

We assume you have the three Java applications running (Foo server, Bar server, and the client). Now notice what happens if you stop the Foo server:

```
INFO Response Hello from Bar server on port 8082
ERROR Failed delivery for (MessageId: ID-davsclaus-pro-
61686-1492951804947
-0-13 on ExchangeId: ID-davsclaus-air-61686-1492951804947-
0-14).
Exhausted after delivery attempt: 1 caught:
org.apache.http.conn.HttpHostConnectException: Connect to
localhost:8081
[localhost/127.0.0.1, localhost/0:0:0:0:0:0:0:1] failed:
Connection refused (Connection refused)
INFO Response Hello from Bar server on port 8082
```

Every second call to the service will fail with an exception because the client can't connect to localhost:8081, which is the physical location of the Foo server. But the service hosted on the Bar server is working, so you can see a successful response when it's called. What can you do to resolve this problem? That's a good question, and there are several solutions. You could, for example, configure Camel's error handler to retry calling the service by configuring the error handler as follows:

```
errorHandler(defaultErrorHandler().maximumRedeliveries(3));
```

Running the client again when the Foo server is unavailable will output responses only from the Bar server:

```
INFO Response Hello from Bar server on port 8082
INFO Response Hello from Bar server on port 8082
INFO Response Hello from Bar server on port 8082
```

And when you start Foo server again, the client will go back to load balance among the two available servers:

```
INFO Response Hello from Bar server on port 8082
INFO Response Hello from Foo server on port 8081
INFO Response Hello from Bar server on port 8082
INFO Response Hello from Foo server on port 8081
```

Why did Camel's error handler fix this problem? When the Service Call is retried (a redelivery is performed by Camel); the service election process starts all over again, according to the algorithm in [figure 17.8](#). And because the two static servers are chosen in a round-robin fashion, the redelivery attempt will select the Bar server this time, which succeeds. To see this, you can configure Camel's error handler to log at WARN level when an attempt fails:

```
errorHandler(defaultErrorHandler().maximumRedeliveries(3)
    .retryAttemptedLogLevel(LoggingLevel.WARN));
```

Then you can see that each attempt to call the Foo server (localhost:8081) fails:

```
INFO Response Hello from Bar server on port 8082
WARN Failed delivery for (MessageId: ID-davsclaus-pro-
61808-1492952557802
-0-1 on ExchangeId: ID-davsclaus-air-61808-1492952557802-0-
2). On delivery
attempt: 0 caught:
org.apache.http.conn.HttpHostConnectException: Connect
to localhost:8081 [localhost/127.0.0.1,
localhost/0:0:0:0:0:0:1] failed:
Connection refused (Connection refused)
INFO Response Hello from Bar server on port 8082
```

Another approach to resolve this problem is to wrap the Service Call EIP with a Circuit Breaker EIP using Hystrix. Hystrix was covered in chapter 7, section 7.4.3.

Health checks

In a distributed system, you're advised to let your services be observable. For example, the services should provide health status so the platform or service registry can

orchestrate this and periodically perform health checks to keep track of the services. The service registry can then filter out unhealthy services, which allows the Service Call EIP to choose only among healthy services. Camel's Service Call EIP allows you to plug in custom providers, such as Consul and Ribbon, that can perform health checks. Chapter 18 covers distributed systems and the importance of health checks.

The Service Call EIP can be configured on different levels. So far, the configuration has been embedded directly within the route. This can become verbose and tedious if you have several service calls in your Camel routes. Instead, you can configure Service Call outside the routes.

17.7.4 CONFIGURING SERVICE CALL EIP

When using the Service Call EIP, it's advised to configure this globally. This ensures that the configuration is done once and is reused by every Service Call EIP in use.

When using Java DSL, you create an instance of `ServiceCallConfigurationDefinition` that's used for the configuration. The example in [listing 17.7](#) can be rewritten, as shown in the following listing.

[Listing 17.9](#) Global configuration of Service Call EIP using Java DSL

```
public class MyStaticRouteGlobal extends RouteBuilder {  
    public void configure() throws Exception {  
        ServiceCallConfigurationDefinition global = ①
```

①

Creates instance of `ServiceCallConfigurationDefinition`

```
    new ServiceCallConfigurationDefinition(); ①
```

```
global.component("http4") ②
```

②

Configures Service Call EIP

```
.staticServiceDiscovery() ②  
    .servers("hello-service@localhost:8081," ②  
            + "hello-service@localhost:8082"); ②
```

```
getContext().setServiceCallConfiguration(global); ③
```

③

Sets global Service Call EIP configuration

```
from("timer:trigger?period=2000")  
    .serviceCall("hello-service/camel/hello") ④
```

④

Calls the service

```
        .log("Response ${body}");  
    }  
}
```

At the top of the `configure` method, we set up the global configuration of the Service Call EIP. At first, we create an instance of `ServiceCallConfigurationDefinition` ①, which is the base for configuring ② the EIP. After the configuration is done, it must be registered in `camelContext`, which is done using the setter method ③.

The Camel route is now much less verbose, and the Service Call EIP is as simple as it can get with just one line of code ④:

```
serviceCall("hello-service/camel/hello")
```

The equivalent example using XML DSL is shown in the following listing.

[Listing 17.10 Global configuration of Service Call EIP using](#)

XML DSL

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

  <defaultServiceCallConfiguration component="http4"> ❶

    ❶
    Configures Service Call EIP

    <staticServiceDiscovery> ❶
      <servers> ❶
        hello-service@localhost:8081,hello-
      service@localhost:8082 ❶
        <servers> ❶
      </staticServiceDiscovery> ❶
    </defaultServiceCallConfiguration> ❶

    <route>
      <from uri="timer:trigger?period=2000"/>
      <serviceCall name="hello-service/camel/hello"/> ❷

    ❷
    Calls the service

    <log message="Response ${body}"/>
  </route>

</camelContext>
```

In XML DSL, you use `<defaultServiceCallConfiguration>` ❶ to configure the default Service Call EIP configuration. As in Java DSL, the Camel route is much simpler; calling the service is only one line of code ❷:

```
<serviceCall name="hello-service/camel/hello"/>
```

RUNNING THE EXAMPLE

The accompanying source code contains these two clients using global configuration, which you can try by running these Java applications from separate command shells. Running the Foo

server:

```
cd chapter17/cluster-servicecall/foo-server  
mvn spring-boot:run
```

Running the Bar server:

```
cd chapter17/cluster-servicecall/bar-server  
mvn spring-boot:run
```

Running the client:

```
cd chapter17/cluster-servicecall/client-static  
mvn camel:run -P global
```

Or the XML DSL version of the client:

```
cd chapter17/cluster-servicecall/client-static-xml  
mvn camel:run -P global
```

Multiple Service Call configurations

Camel allows you to configure multiple Service Calls, with each configuration associated with a unique name. You can then refer to which configuration to use from the Service Call EIP in your Camel routes. You can find information on how to do this from the Camel documentation on GitHub:

<https://github.com/apache/camel/blob/master/camel-core/src/main/docs/eips/serviceCall-eip.adoc>.

Next we'll talk about how the Service Call EIP allows you to specify the Camel endpoint URI to be used when calling the service.

17.7.5 SERVICE CALL URI TEMPLATING

A key feature of Apache Camel is endpoint URIs, which make it easy to use the many Camel components. When the Service Call EIP calls a service, it constructs a Camel endpoint URI to use.

These URIs support URI templating.

A picture is worth a thousand words, so let's try with a table instead; see table 17.1.

Table 17.1 Service Call URI template examples

Name	Resolved URI
hello-service	http4:host:port
hello-service/camel/hello	http4:host:port/camel/hello
hello-service/camel/hello?id=123	http4:host:port/camel/hello?id=123

The first row is the most basic example, with only the logical name of the service. The logical name is then replaced with the chosen physical address of the service in the form host:port. With the static server list example, this would resolve as either http4:localhost:8081 or http4:localhost:8082. The middle row is the name used in the example. As you can see, this allows you to include context-path in the resolved URI. The last row shows how you can even include query parameters.

ADVANCED URI TEMPLATING

If you want full control of the way the URI is resolved, you can specify the URI yourself. You may need to do this when calling services that aren't using HTTP/REST transport. Then you may find yourself using some of the other Camel components, which may require constructing the URIs in a certain way.

But we can still show you how to construct a URI using HTTP transport. For example, you can hardcode the URI to use another Camel component such as undertow:

```
.serviceCall("hello-service",
            "undertow:http://hello-service/camel/hello?
id=123")
```

And in XML DSL:

```
<serviceCall name="hello-service"
```

```
        uri="undertow:http://hello-
service/camel/hello?id=123"/>
```

Notice that you separate the service name and the URI:

Name	URI template
hello-service	undertow: http://hello-service /camel/hello?id=123

Pay attention to the highlights in the table. This is the name of the service, which must be represented in the URI template. Camel will then replace the service name with the physical address of the chosen server in the format host:port—for example:

```
undertow:http://localhost:8081/camel/hello?id=123
```

You can even specify the exact position of the hostname and port number individually. For instance, the same example can be defined using name.host and name.port in the syntax as highlighted:

```
undertow:http://hello-service.host:hello-
service.port/camel/hello?id=123
```

The last example we'll cover uses Spring Boot Cloud and Consul as the service registry.

17.7.6 SERVICE CALL USING SPRING BOOT CLOUD AND CONSUL

HashiCorp Consul can be used as a dynamic service registry in a distributed system. Spring Boot also provides support for Consul, which can be integrated easily with Camel. The example is located in the chapter17/cluster-servicecall/client-consul directory.

In the Maven pom.xml, we've added the Spring Boot starter for Consul:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-
```

```
discovery</artifactId>
</dependency>
```

And the Camel starters:

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

Then enable service discovery by adding the `@EnableDiscoveryClient` annotation to the application class (the class with the `@SpringBootApplication` annotation).

Next you configure Consul in the Spring Boot `application.properties` file:

```
spring.cloud.consul.discovery.enabled=true
spring.cloud.consul.discovery.server-list-query-tags[hello-service] = camel
```

Note the last line, where you map the name of the service to Consul service tags. This is important because this is how Consul knows how services are grouped together.

The Camel route has no need for any Service Call configuration, which makes this easy to use:

```
from("timer:trigger?period=2000")
    .serviceCall("hello-service/camel/hello")
    .log("Response ${body}");
```

That's all you need to develop on the client side. But you may wonder how Consul knows where the services are physically located. When starting Consul, you can specify the services in a JSON file. For example, the Foo and Bar servers are specified as follows:

```
{
  "services": [
    {"id": "hello-foo",
```

```
        "name": "hello-service",
        "tags": ["camel"],
        "address": "localhost",
        "port": 8081
    }, {
        "id": "hello-bar",
        "name": "hello-service",
        "tags": ["camel"],
        "address": "localhost",
        "port": 8082
    }]
}
```

You can run this example by following the instructions in the readme file from the chapter17/cluster-servicecall directory. The example uses Docker to run Consul, and you should pay attention on how to do this.

SERVICE CALL WITH FAILOVER

If you run this example and, for example, stop one of the servers, you'll see the same problem as in section 17.7.3. The client will fail with an exception. We can't say this enough: when you build distributed applications, *design for failure*. You can resolve this problem the same way as previously suggested. But if you're using Consul in your organization, you may want to use its support for health checks so Consul can observe your applications and keep its service registry up-to-date with the health state of all services.

This requires you to run a Consul agent on each node, which becomes responsible for running the health checks and reporting back to the Consul cluster. In addition, you need to specify how to perform health checks for your services, which you can do using techniques such as an HTTP call or a shell script. This topic is beyond the scope of this book, but you can consult the Consul documentation. What we want to say is that this is all much easier with container-based platforms such as Kubernetes. All of this is baked into the platform, and you have health checks and service discovery out of the box.

That brings us to the end of our coverage of using clustered Camel with traditional Java technologies.

17.8 Summary and best practices

This chapter was a tour of the various ways of creating clustering with Camel using traditional solutions. For example, we talked about using HTTP load balancers, which is a well-established solution. Clustering Camel routes is done using numerous Camel components that integrate with existing proven cluster solutions.

You may be wondering why a chapter on clustering Camel is so far back in this book. One reason is that clustering is a hard topic, and there are many other topics we wanted to cover first. Another reason is related to Camel's background. When Camel was created, it was created as a small integration framework that could be embedded inside existing solutions. For example, combining the following individual Apache projects gives you a powerful solution that supports clustering as well:

- *Apache Camel*—Integration framework with EIPs and components
- *Apache CXF*—Web and RESTful services
- *Apache ActiveMQ*—JMS messaging and clustering
- *Apache Karaf*—OSGi-based application server
- *Apache ServiceMix*—Umbrella project that embeds all the preceding list items into a Camel-based integration platform

In light of this, Camel wasn't intended to be an all-in-one, kitchen-sink solution. Camel was focused on being a small integration framework and left clustering and application management to ActiveMQ, Karaf, and ServiceMix. The story today is different. We're moving toward a microservice architecture approach, so Camel must do clustering well without relying on other Apache projects.

In this chapter, you've seen a variety of ways to cluster Camel.

As always, Camel is flexible; it would be possible to build your own Camel component or route policy to add clustering capabilities to Camel that aren't provided out of the box.

The noteworthy takeaways from this chapter are as follows:

- *Clustered components*—Clustering in Camel is primarily provided by Camel components. When it comes to clustering, Camel is just a library, and you should look at what clustering solutions you have that you can use with Camel. For example, you can use HTTP load balancers, JMS message brokers, or clustered caches, to name just a few.
- *Clustered routes*—If you need to cluster Camel routes, techniques are available in either active/passive or active/active mode. You can use several Camel components to integrate with a clustered solution, such as Hazelcast, Infinispan, Consul, or ZooKeeper.
- *Clustered messaging is hard*—Don't underestimate the complexity of distributed messaging. There's no easy solution that works in all situations. We encourage you to conduct comprehensive analysis and read up on relevant material (books, articles, and other resources).
- *Use Service Call*—If your Camel applications must call services that run in a distributed system, consider using the Service Call EIP. This EIP is flexible and allows you to separate configuration from the service call; you can migrate from using, for example, Consul to Kubernetes easily. You can also build custom strategies to filter and choose from among the services. For example, you can plug in logic to choose services using the canary deployment principle, or to prioritize calls according to plans, or SLAs.

The Camel team has begun implementing new clustering and health check APIs and services. This work is an ongoing effort and was first released as a technical preview in Camel 2.20. Because the work is under active development and is changing, it was too early to cover in this book. If you want to follow this

work, we recommend reading "A camel running in the clouds" by Luca Burgazzoli (creator of this work) at his blog:
<https://lburgazzoli.github.io>.

We'll continue our clustering endeavor when we move on to the world of containers. Chapter 18 will introduce you to container-based technologies such as Docker and Kubernetes. Let's jump on the horse—er, Camel—and ride the wave of new awesomeness of the promised land that containers are shipped from. Okay, get down from your high horse—ugh, darn, not again—we meant Camel. But speaking more seriously, *container* isn't yet another buzzword that will be gone in a few years. Containers are a game changer, so let's see how well Camel rides in containers.

18

Microservices with Docker and Kubernetes

This chapter covers

- Running Camel on Docker
- Getting started with Kubernetes
- Running and debugging Camel on Kubernetes
- Understanding essential Kubernetes concepts
- Building resilient microservices on Kubernetes
- Testing Camel microservices on Kubernetes
- Introducing fabric8, Kubernetes Helm, and OpenShift

If you've been developing Java applications for many years, the transition to running them on the cloud is a big leap. You not only need to learn and master new concepts and primitives from the container-based world, you also make an architectural change from a monolithic to a microservice style.

As a developer, this can be a daunting mountain to climb, so in this chapter we'll help you focus on one thing at a time and climb that mountain step by step. We'll be in the developer role and show you how to develop, build, and run Camel microservices on Docker and Kubernetes all running locally on your computer.

You aren't required to use any on-premises cluster infrastructure or sign up with an online cloud provider. As developers, we feel comfortable if we have control and can do it all from our own computers. You don't have to worry about whether your computer is powerful enough, because what we'll do can run on any reasonable computer. At the time of this writing, the author of this chapter is using a four-year-old MacBook Air equipped with only 8 GB of memory and a mediocre CPU.

Docker came out a number of years ago as an elegant solution for packaging and running applications. Section 18.1 starts with Docker; we'll show you how to take any Java project and build it as Docker images and run it using Docker containers.

Docker is so popular that you may have heard that it solves all your problems. Although Docker provides a great portable container format for shipping applications between environments and allows you to run these applications anywhere, it's just one part of a bigger puzzle. When you move beyond the single-container phase and run your applications in a cluster of nodes, you need something more. You need powerful clustering facilities such as service discovery, load balancing, fault-tolerance, container scheduling, rolling deployment to minimize downtime, mounting persistent volumes, and configuration management. That's where Kubernetes enters the stage. It has all of that, and it's the third-generation container platform from Google, which brings tons of wisdom about running containers reliably at scale.

When getting started with microservices built and deployed in Docker and managed by Kubernetes, it helps to be running a local environment that's used for development purposes. In section 18.2, you'll learn how to set up a local Kubernetes cluster. In section 18.3, you'll get down to action and run Camel microservices in the local Kubernetes cluster.

Section 18.4 covers essential Kubernetes concepts that you should learn in order to be familiar with container-based

platforms. These concepts are universal, so if you find yourself using an alternative cloud or container platform, you can apply the information from this section to your situation as well.

A distributed system such as a microservices architecture will affect developers who now must even more rigorously build their applications with fault-tolerance in mind. Section 18.5 shows what can go wrong when you don't do so. We'll play dirty and cause chaos in the local cluster by killing containers so you can see what happens. With this insight and practical experience, you'll learn how to develop your Camel applications to be resilient and fault-tolerant. You'll also learn to use what Kubernetes brings to the table in this matter via support for self-healing. Section 18.6 covers the basics of writing and running unit tests with Kubernetes.

We end this chapter by stepping beyond Kubernetes to briefly cover three projects: fabric8, Helm, and OpenShift, which all bring different value to Kubernetes users. Let's start with the little blue whale known as Docker.

18.1 Getting started with Camel on Docker

Docker, Docker, Docker. Unless you've been living under a rock for the last couple of years, you've likely heard of Docker. You may even have become tired of hearing that it's the best thing since sliced bread and that it'll change everything.

We've certainly had our dose of Docker and therefore won't spend too much time introducing you to Docker and explaining its installation and use. Docker is a key component in Kubernetes and therefore also in the way we build, package, and deploy our applications in clustered platforms—maybe already today and certainly in the near future. With this in mind, we think it's important to show you how to build and run your Java applications (and Camel) with Docker and Kubernetes.

We'll do this by developing two small microservices that use Camel and run on two popular microservices frameworks:

Spring Boot and WildFly Swarm. The microservices we'll build, deploy, and run are illustrated in [figure 18.1](#).

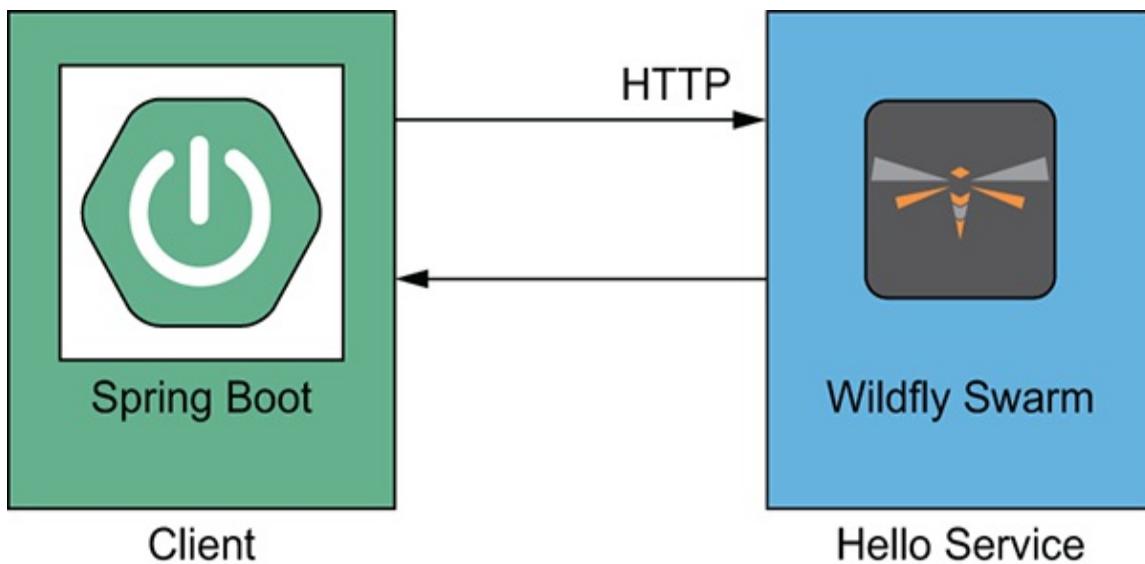


Figure 18.1 The microservice on the left runs on Spring Boot and is a client application that periodically calls the Hello service and logs its response. The microservice on the right runs using WildFly Swarm and hosts a Hello service returning a canned response.

This section and the following two sections will take you on a journey from developing the microservices and running them standalone, to using Docker and finally using Kubernetes. Metaphorically, we're climbing up a container mountain. The steps to the top of the Kubernetes mountain are as follows:

1. Develop the two microservices
2. Run the microservices standalone
3. Build the microservices as Docker images
4. Run the microservices using Docker

In the following sections (18.2 and 18.3) you'll continue using Kubernetes:

1. Install and run Kubernetes
2. Run the microservices using Kubernetes

Put on your climbing gear, because here goes.

18.1.1 BUILDING AND RUNNING CAMEL MICROSERVICES LOCALLY

Chapter 7 covered building Camel microservices with WildFly Swarm and Spring Boot. To quickly build the microservices for this chapter, you'll use the same approach; you'll visit the websites <http://start.spring.io> and <http://wildfly-swarm.io/generator>.

In the interest of not repeating ourselves, the book's source code contains the ready-built microservices in the chapter18/standalone directory. The highlights are as follows.

WildFly Swarm includes a Camel route that can be written as one line:

```
from("undertow:http://localhost:8080/").bean(hello);
```

The called bean then returns a canned response that includes the hostname:

```
public String sayHello() throws Exception {  
    return "Swarm says hello from " +  
    InetAddressUtil.getLocalHostName();  
}
```

The Spring Boot client is simple as well, as it's just a Camel route:

```
from("timer:foo?period=2000")  
    .to("http4://localhost:8080")  
    .log("${body}");
```

RUNNING THE EXAMPLE LOCALLY

You can now run this example locally by first starting the WildFly Swarm application:

```
cd hello-swarm  
mvn wildfly-swarm:run
```

Then from another shell, start the client:

```
cd client-spring
```

```
mvn spring-boot:run
```

The client will then call the Hello service every other second and log the response:

```
[0 - timer://foo] route1 : Swarm says hello from  
davsclaus.air  
[0 - timer://foo] route1 : Swarm says hello from  
davsclaus.air
```

When you run multiple Java JVMs on your computer, you may have had the problem that one JVM can't start because it wants to use a TCP port that's already taken by another running JVM. We'd have this problem in this example if we didn't avoid this deliberately.

RUNNING SPRING BOOT WITHOUT AN EMBEDDED HTTP SERVER

Spring Boot makes it easy to embed an HTTP server using the `spring-boot-starter-web` dependency. But for simple standalone clients, this may not be needed, so you can use the `spring-boot-starter` dependency instead. Ironically, doing so makes it a bit harder to keep Spring Boot running. But Camel makes this easy by turning on the run controller in the `application.properties` file:

```
camel.springboot.main-run-controller=true
```

Spring Boot will now keep running until the JVM is terminated.

Spring Boot with embedded HTTP server and health checks

Running Spring Boot without an embedded HTTP server is recommended in only some situations, such as for development purposes or for running as tiny a JVM as possible without having to include an HTTP server. On the other hand, when running Spring Boot in a cloud platform

such as Kubernetes, it's recommended to include an embedded HTTP server with Spring Boot to more easily facilitate health checks, which help to achieve robust and highly available applications.

You're now ready to climb the next step up the Kubernetes mountain.

18.1.2 BUILDING AND RUNNING CAMEL MICROSERVICES USING DOCKER

Docker and Kubernetes run applications as containers that are loaded from Docker images. How do you build a Docker image? That's a good question; building a Docker image is easy and hard at the same time.

It's easy because the Docker images are defined using a Dockerfile, which you can easily write on your own. A Dockerfile is just a text file with that very same name. For example, the Dockerfile you'll use to run the Spring Boot client microservice contains just three lines of text:

```
FROM openjdk:latest
COPY maven /maven/
CMD java -jar maven/spring-docker-2.0.0.jar
```

A Docker image is a compressed TAR file that includes the Dockerfile in the root alongside other files you want to include in the image. The Spring Boot Docker image consists of only two files:

```
maven/spring-docker-2.0.0.jar
Dockerfile
```

This sounds easy, so why is building a Docker image also hard? Well, if the IT industry were simple, you wouldn't need to buy and read a 900-page book to know everything about Apache Camel. And with Docker images, things get more complicated as you need to set up user permissions, set environment variables, mount volumes, map network ports, run commands, and much

more.

Luckily, some great tooling can assist you, especially for Maven-based Java projects. To build a Docker image, you should use the Docker Maven plugin from fabric8.

FABRIC8 DOCKER MAVEN PLUGIN

The fabric8 project has great tooling for Java developers to work with Docker and Kubernetes. To build Docker images, you'll use docker-maven-plugin by adding the plugin to the two microservices. The Spring Boot client is the easiest, as all you have to do is add the code in the following listing.

[Listing 18.1](#) Building a Docker image for the Spring Boot microservice

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.23.0</version>
  <configuration>
    <images>
      <image>
        <name>camelinaction/spring-
docker:latest</name> 1
    
```

1

Name of the Docker image

```
    <build>
      <from>openjdk:latest</from> 2
    
```

2

Base Docker image

```
    <assembly>
      <descriptorRef>artifact</descriptorRef> 3
    
```

3

What to include in the Docker image

```
</assembly>
<cmd>
  java -jar maven/${project.artifactId}-
${project.version}.jar ④
```

④

Command to start the application

```
</cmd>
  </build>
  </image>
  </images>
</configuration>
</plugin>
```

A Docker image must have a name in this syntax:

organization/name:version

In this example, we use `camelinaction` as the organization and `spring-docker` as the name, and `latest` as the version ①.

Using placeholders in a Docker image name

The Docker Maven plugin supports defining the Docker image name with placeholders, as shown here:

```
<name>%g/%a:%l</name>
```

This uses the Maven group ID as the organization, Maven artifact ID as the name, and the Maven version as the version (unless the version is SNAPSHOT, and then latest is used). You can find more details in the docker-maven-plugin documentation: <https://dmp.fabric8.io/#image-configuration>.

Docker images are layered, and you need to specify the parent image, which in our example must include Java, so you choose latest with openjdk ②. The assembly configures which files to include in the Docker image. Because you’re using Spring Boot as a fat JAR, you need to include only one file, which is the artifact built ③. And finally, you specify how to run the application in the run configuration ④. You can now build the Docker image by running the following:

```
cd chapter18/docker/client-spring  
mvn install docker:build
```

This builds the Docker image and pushes it to the local Docker image repository. You should be able to see the built image by running the `docker images` command:

```
$ docker images  
REPOSITORY          TAG      IMAGE ID      CREATED  
SIZE  
camelinaction/spring-docker  latest   a0ba61b85c6f  8  
minutes ago    657 MB
```

Why are Docker images so large?

If you look at the output from the `docker images` command in the preceding example, it shows that the `camelinaction/spring-docker` image is 657 MB. Docker images are layered, and an image includes all the parent layers and their dependencies. In this case, the base image is `openjdk`, which includes a Linux distribution and Java that end up taking up the vast majority of the size.

There’s no point in running the Spring Boot client yet, because you need to build and run the WildFly Swarm microservice first.

18.1.3 BUILDING A DOCKER IMAGE USING THE DOCKER MAVEN PLUGIN

You'll also use docker-maven-plugin to build the WildFly Swarm microservice. This requires a bit more work than with Spring Boot, as shown in the following listing.

[Listing 18.2](#) Building a Docker image for the WildFly Swarm microservice

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.23.0</version>
  <configuration>
    <images>
      <image>
        <name>camelinaction/helloswarm-docker</name>
```

1

1

Name of Docker image

```
    <build>
      <from>openjdk:latest</from>
```

2

2

Base Docker image

```
    <assembly>
      <inline>
```

3

3

What to include in the Docker image

```
      <fileSets>          3
        <fileSet>          3
          <directory>target</directory>      3
          <outputDirectory>.
        </outputDirectory>    3
        <includes>          3
          <include>*-swarm.jar</include>  3
        </includes>          3
      </fileSet>          3
    </fileSets>          3
```

```
        </inline>      ③  
    </assembly>  
<cmd>  
    java -jar maven/${project.artifactId}-${project.version}-  
swarm.jar ④
```

④

Command to start the application

```
</cmd>  
    </build>  
    <run>  
        <ports>  
            <port>8080:8080</port> ⑤
```

⑤

Network port mappings

```
        </ports>  
    </run>  
    </image>  
    </images>  
  </configuration>  
</plugin>
```

As with Spring Boot, you define the name of your Docker image ①. Because you're using Java, you start off with the openjdk image ②. When using WildFly Swarm with Docker, you want to run it as a fat JAR (or fat WAR). Because the wildfly-swarm Maven plugin generates the fat JAR with `-swarm` as its suffix, you need extra configuration to work around that. What you can do is inline the assembly and use file sets to include the correct fat JAR ③. The fat JAR can then easily be run from Java using `java -jar` as specified in the `run` command ④. Since the WildFly Swarm microservice is hosting an HTTP service, you need to expose this in the Docker port mappings ⑤. The syntax is `external-port:internal-port`. People usually map port numbers using the same internal and external port numbers, so you typically see `2181:2181`, `8080:8080`, and so forth. With this port mapping, you can let the Spring Boot microservice call the Hello

service within the WildFly Swarm application on port 8080.

The Docker image can be built by running the following:

```
cd chapter18/docker/hello-swarm  
mvn install docker:build
```

When the image is built, you can see it from the local Docker image repository:

```
$ docker images  
REPOSITORY           TAG      IMAGE ID  
CREATED      SIZE  
camelinaction/helloswarm-docker latest   a0d46b208134  7  
minutes ago  755 MB
```

Okay, you should now be ready to run this example using Docker.

18.1.4 RUNNING JAVA MICROSERVICES ON DOCKER

You first need to run the WildFly Swarm microservice that hosts the HTTP service the client will be calling. You can run this from a shell:

```
docker run -it -p 8080:8080 camelinaction/helloswarm-  
docker:latest
```

When running Docker applications, you can run them in the background or foreground. Well, that's not entirely true. You can run the application in interactive mode using `-it`, which means the shell will tail the logs and keep the container running until the shell is terminated with Ctrl-C. This makes it nice and easy for developers to run something and quickly stop, and then do code changes, rebuild, and run again.

Notice that you also specify a port mapping with `-p 8080:8080`. You might think you must do this because the Spring Boot client has to access the HTTP service on this port. But that's not exactly why. Using `-p 8080:8080` creates a port forwarding between the host operating system and Docker. This allows you to call the HTTP service from a web browser on `localhost:8080` as if the

application were running natively on your host. Open a web browser and enter the following URL:

```
http://localhost:8080
```

You should see a response from WildFly Swarm such as this:

```
Swarm says hello from f1c5a96e250c
```

Now that WildFly Swarm is up and running, you're ready to start the Spring Boot client. You can do this by running the following Docker command:

```
docker run -it camelinaaction/spring-docker:latest
```

The application starts up and calls the HTTP service, and all is good. Hey, wait, what's happening? The application fails, and a stacktrace is logged; why is that?

Message History

```
-----  
RouteId ProcessorId Processor  
Elapsed (ms)  
[route1] [route1] [timer://foo?period=2000] [  
69]  
[route1] [to1 ] [http4://localhost:8080] [  
63]
```

Stacktrace

```
-----  
org.apache.http.conn.HttpHostConnectException: Connect to  
localhost:8080  
[localhost/127.0.0.1, localhost/0:0:0:0:0:0:1] failed:  
Connection refused  
(Connection refused)
```

The client can't connect to localhost:8080, but you just tried that from a web browser. You called <http://localhost:8080> and got back a response from WildFly Swarm. Why doesn't it work for the Spring Boot application?

The reason is related to the way Docker runs applications in isolated containers when each container is assigned its own IP

address, as pictured in [figure 18.2](#).

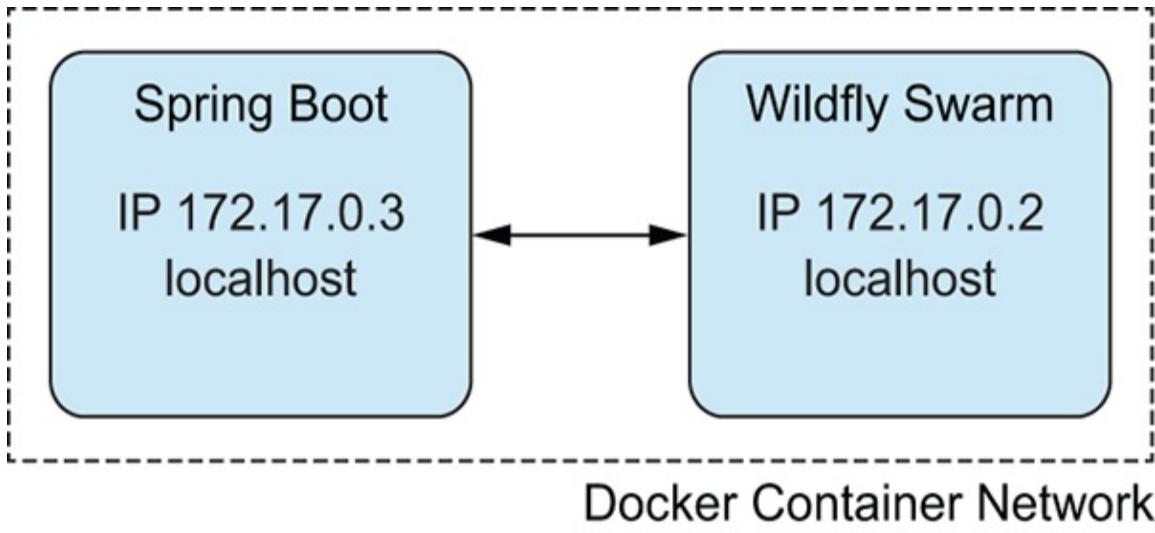


Figure 18.2 Each Docker container is isolated and has its own unique assigned IP address. The applications running inside the containers can't communicate with each other using localhost, but must use the IP address instead.

COMMUNICATING BETWEEN DOCKER CONTAINERS

Each Docker container that runs on the same Docker host runs in the same isolated Docker network. Each container gets a unique IP address assigned, and therefore the applications can connect over the network using this IP number. In [figure 18.2](#), you can see that the Spring Boot container was assigned 172.17.0.3, and WildFly Swarm was assigned 172.17.0.2.

To fix the problem with running the Spring Boot client, you need to change the Camel route to use 172.17.0.2 instead of localhost:

```
.to("http4://172.17.0.2:8080")
```

We don't like hardcoding an IP address, so let's use a property placeholder:

```
.to("http4://{{swarm.ip}}:8080")
```

You can then specify the address in the Spring Boot application.properties file:

```
swarm.ip=172.17.0.2
```

That's a better practice.

Okay, but how do you know the IP address that Docker assigned? Frankly, you don't know. That has a lot to do with the way you run Docker and the way you may have set up Docker Container Networking. But by default, Docker assigns IP addresses from 172.17.0.2 upward. When you started WildFly Swarm first, it was assigned the first free number, which is 172.17.0.2. But then to be sure, you need to inspect the running container to find its IP address, which you can do as follows:

```
$ docker ps
CONTAINER ID        IMAGE
f1c5a96e250c      camelinaaction/helloswarm-docker:latest
```

To find the container ID of the running container, which you can then inspect, you run this:

```
docker inspect f1c5a96e250c
```

This outputs a lot of information, but within that you can find the IP address:

```
"IPAddress": "172.17.0.2",
```

Now that you have the correct IP address and have updated the Spring Boot source code, you can rebuild and run the application again:

```
mvn clean install docker:build
```

Then run the application using Docker:

```
docker run -it camelinaaction/spring-docker:latest
```

Because you're using the correct IP address, the console logs the response from WildFly Swarm:

```
route1 : Swarm says hello from f1c5a96e250c
route1 : Swarm says hello from f1c5a96e250c
route1 : Swarm says hello from f1c5a96e250c
```

Is there something wrong with the log? Why does it log such a weird name, f1c5a96e250c? There's nothing wrong with the log. When Docker runs a container, that container is assigned a unique ID that becomes the hostname.

To stop the application, you can press Ctrl-C.

Java inside Docker: what you must know to not fail

Running Java inside Docker containers does bring new problems. For example, Java may see that it's running on an operating system that has eight CPU cores, but in reality the Docker container has been resource-limited to one CPU core. Rafael Benevides posted a blog with good details:

<https://developers.redhat.com/blog/2017/03/14/java-inside-docker/>.

LOOKING INSIDE A RUNNING CONTAINER

When running Docker containers, you need from time to time to be able to explore inside to see what's going on. You can start a new process inside the existing running container, even a shell such as bash if bash is provided by the Docker image (typically, bash or sh is provided from a base image).

For example, to look inside the WildFly Swarm container, you need to find its container ID and then start the bash shell using docker exec as highlighted here:

```
$ docker ps
CONTAINER ID        IMAGE
ccb99db22935      camelinaction/spring-docker:latest
f1c5a96e250c      camelinaction/helloswarm-docker:latest
$ docker exec -it f1c5a96e250c bash
root@f1c5a96e250c:/# ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START
```

```
TIME COMMAND
root      1 0.0 0.0      4336    760 ?          Ss+ 15:17
0:00 /bin/sh
 -c java -jar maven/helloswarm-docker-2.0.0-swarm.jar
root      5 0.4 20.3 3090760 416828 ?          S1+ 15:17
0:29 java
 -jar maven/helloswarm-docker-2.0.0-swarm.jar
```

To quit, you run the `exit` command.

This concludes our coverage of running Camel on Docker. We want to show you the transition from running Camel standalone, via Docker, to running Camel on a real container platform such as Kubernetes.

18.2 Getting started with Kubernetes

This section will teach you the basic climbing skills so you'll learn the basics before we let you attempt climbing the Kubernetes mountains. At first, we'll walk you through installing and running a local Kubernetes cluster. We'll show you two ways to deploy and run applications in Kubernetes. In this section, we'll use the Kubernetes command-line tooling so you're aware of its existence and can gain knowledge of how to use it. The command-line tool works for any kind of application you want to run, not only Java-based applications. In section 18.3, we'll take you back to Java and Camel and use the fabric8 Maven plugin, which makes it much easier for Java developers to build, run, and debug applications in Kubernetes.

After some practice with using Kubernetes, you'll learn in section 18.4 the theories behind the most important concepts and principles of Kubernetes. We could have done this in reverse order (theory before practice), but we want you to have fun, and we think you can best learn Kubernetes by seeing something running in action first.

Kubernetes (and Docker) are native Linux technologies, so they must run in a Linux operating system. You'll need to use something called Minikube (Minikube is a local Kubernetes

cluster intended for development purposes). Minikube makes it easy to run a local single-node Kubernetes cluster on a Windows, Mac, or Linux computer.

Let's get started and install Minikube.

18.2.1 INSTALLING MINIKUBE

Minikube requires a virtualization driver to be preinstalled. The driver varies depending on your operating system. Mac users should install the xhyve driver. Windows users should use Hyper-V. Linux users can use KVM. You can find information and links for installing these drivers on the Minikube website: <https://github.com/kubernetes/minikube>. From this point forward, we assume that you've installed a virtualization driver.

To install and run Minikube, you can follow the instructions from the Minikube website. For example, to install on a Mac:

```
$ curl -Lo minikube
https://storage.googleapis.com/minikube/releases/
v0.24.1/minikube-darwin-amd64 && chmod +x minikube &&
sudo mv minikube /usr/local/bin/
```

After Minikube has been installed, you should be able to run the `version` command:

```
$ minikube version
minikube version: v0.24.1
```

Minikube is now ready to be started.

18.2.2 STARTING MINIKUBE

When starting Minikube for the first time, you can provide parameters to specify the amount of memory, CPU, and disk space to allocate. The defaults are reasonable, but for our little example you don't need as much resources, so start with the following parameters:

```
minikube start --cpus 2 --memory 2048 --disk-size 10g --vm-
driver xhyve
```

The last parameter is important; it specifies which VM driver to use (see Minikube documentation for details). After the installation is complete, you can get the status of Minikube:

```
$ minikube status  
minikubeVM: Running  
localkube: Running  
kubectl: Correctly Configured: pointing to minikube-vm at  
192.168.64.2
```

This means the local Kubernetes cluster is up and running.

Starting Minikube from scratch

The good thing about Minikube is that you shouldn't fear that it'll take over your computer. At any time, you can always delete and start a new cluster. For example, if a cluster is running, you first stop it:

```
minikube stop
```

Then you can delete the cluster:

```
minikube delete
```

And then you can run start again, to create a new cluster:

```
minikube start --cpus 2 --memory 2048 --disk-size 10g  
--vm-driver=xhyve
```

If you find your Minikube installation broken, you can always delete the `~/.minikube` directory and start again, as Minikube will then re-download needed Docker images and reconfigure itself.

But how do you interact with the cluster?

INTERACTING WITH KUBERNETES

Kubernetes provides a command-line tool called `kubectl` that you

use to interact with the cluster. For example, you can deploy applications, scale deployments up or down, and a lot more.

You install kubectl by following the installation guidelines from the Kubernetes website:

<https://kubernetes.io/docs/tasks/tools/install-kubectl/>.

After you've installed kubectl and it's working, you can run the `cluster-info` command:

```
$ kubectl cluster-info  
Kubernetes master is running at https://192.168.64.2:8443
```

A-ha—there's something called a Kubernetes master. We'll get back to that in section 18.4.

Minikube also comes with a web dashboard, and you can obtain the URL as follows:

```
$ minikube dashboard --url  
http://192.168.64.2:30000
```

Let's open a web browser and see this dashboard. You could copy/paste that URL, but instead you can type the command without the `--url` parameter:

```
minikube dashboard
```

The dashboard then opens in your web browser. Spend a few minutes clicking the dashboard, and you'll find out that there's not much to see. That's because no applications are running in the cluster. But something is running in the cluster: Kubernetes runs a core set of services that are part of what's called the *control pane*. Because Kubernetes runs containers, you can use the Docker CLI to see this.

USING DOCKER CLI WITH KUBERNETES

A good idea when working with Kubernetes from the shell is to set up the Docker environment to point to the Kubernetes cluster. You can do this from Minikube:

```
$ minikube docker-env  
export DOCKER_TLS_VERIFY="1"  
export DOCKER_HOST="tcp://192.168.64.2:2376"  
export DOCKER_CERT_PATH="/Users/davsclaus/.minikube/certs"  
export DOCKER_API_VERSION="1.23"  
# Run this command to configure your shell:  
# eval $(minikube docker-env)
```

This outputs the command to run:

```
eval $(minikube docker-env)
```

You can now use the Docker CLI to interact with the Kubernetes cluster. For example, to list all the running Docker containers, you type this:

```
docker ps
```

And to list which Docker images are in the Kubernetes Docker repository, you type this:

```
docker images
```

Starting and stopping Minikube

At any time, you can stop Minikube as follows:

```
minikube stop
```

And then later, you can start Minikube again, and it'll resume where it left off:

```
minikube start
```

After all this, your local Kubernetes cluster is up and running and ready. Now it's time to run your first application in the cluster.

18.3 Running Camel and other applications

in Kubernetes

When you run applications on Kubernetes, they run as containers loaded from Docker images. The information you learned in the previous section about running Camel on Docker is required knowledge for working with Kubernetes.

To deploy an application in Kubernetes, you need two things:

- Docker image
- Kubernetes manifest

The Kubernetes manifest is a file that holds metadata about how to set up and deploy the application. This manifest file can be in either YAML or JSON format and can be written by hand. But when applications become more complex, it's recommended to generate these manifests. You'll see how to do that in the following section. But first let's start from nothing and try to run the WildFly Swarm example you built as a Docker image.

18.3.1 RUNNING APPLICATIONS USING KUBECTL

From a shell, you can run the WildFly Swarm application using the `run` command from `kubectl`. Before you can do this, you need to build the Docker image with the application, and you can do that by running the following commands from a shell (section 18.3.3 dives into more detail about building and deploying to Kubernetes using Maven):

```
eval $(minikube docker-env)
cd chapter18/kubernetes/hello-swarm-docker
mvn package fabric8:build
```

The `fabric8:build` command will build the Docker image and push that to the local Docker repository, which you can list using the Docker command line as follows:

```
$ docker images
REPOSITORY          TAG      IMAGE ID
CREATED
```

```
camelinaction/helloswarm-kubernetes 2.0 34326d1f695e 2  
minutes ago
```

Now you're ready to run the application using the `kubectl` tool:

```
$ kubectl run hello-world --replicas=1 --labels="foo=bar"  
--image=camelinaction/helloswarm-kubernetes:2.0 --  
port=8080
```

This command tells Kubernetes to run a new application with the name `hello-world` with one instance. The application has an associated label with its key-value as `foo=bar`. The application uses the Docker image from the `camelinaction` organization with the name `helloswarm-kubernetes` and `2.0` as the version. The last parameter is used for exposing a network connection on port `8080` to the running container.

kubectl with shell completion

We recommend that you install shell completion for the `kubectl` tool, which allows you to use tab completion. For example, to show the logs of a running pod, you can type the following from the command shell:

```
kubectl log he<TAB>
```

Instead of typing `<TAB>`, you press the Tab key. This completes the name of the pod:

```
kubectl log hello-world-1096927984-m0jpk
```

Otherwise, you'd have to type the name. You can find instructions on installing shell completion at the Kubernetes website: <https://kubernetes.io/docs/tasks/tools/install-kubectl/#enabling-shell-autocompletion>.

This application runs inside Kubernetes as a Docker container, in what Kubernetes calls a *pod* (pods will be covered in section 18.4). To list the running applications, you list the pods using `get`

pods:

```
$ kubectl get pods
NAME                               READY   STATUS    RESTARTS
AGE
hello-world-1096927984-m0jpk     1/1     Running   0
12m
```

Here you can see that one pod is running.

18.3.2 CALLING A SERVICE RUNNING INSIDE A KUBERNETES CLUSTER

Okay, so far so good, but how do you try calling the HTTP service that the example provides? How do you, from your host operating system in a web browser, call an HTTP URL inside the Kubernetes cluster? That is an excellent question. By default, Kubernetes doesn't expose network ports outside the cluster. What you need to do is expose the network port 8080 as a service, which also can be accessed from outside the Kubernetes cluster. This can be done using the expose command:

```
kubectl expose deployment hello-world --type=NodePort --
name=hello-service
```

This command exposes a service on the hello-world deployment of type NodePort. Because you run the Kubernetes cluster using Minikube as a single-node cluster for development purposes, all you need is a way to forward network connections from the Kubernetes master to port 8080 on the running pod. Figure 18.3 illustrates the players involved as you call the service from your web browser.

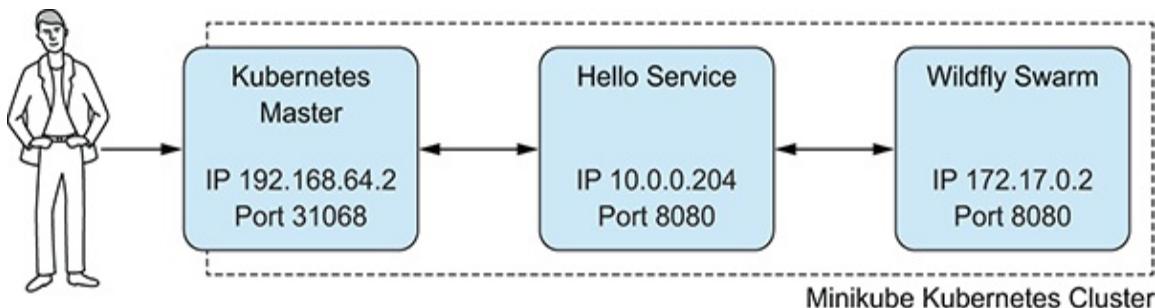


Figure 18.3 A user wants to call a service running inside the Kubernetes cluster.

The service is accessible through node port 31068 on the Kubernetes master, which forwards the traffic to the Hello service running inside the cluster. The Hello service also forwards the traffic to the running pod with the targeted service being called.

Minikube is running locally on your computer as a virtual machine (VM). The VM has been assigned IP number 192.168.64.2, which is reachable from your host operating system. You can communicate with applications running inside the Minikube cluster with this IP number.

In Kubernetes, you use Kubernetes services as a level of indirection between your running pods and clients calling into these pods. (Section 18.4 goes more in depth about the Kubernetes concepts.) In [figure 18.3](#), you can see that a Hello service sits between the Kubernetes master and the pod that runs the WildFly Swarm application. At this time, all you need to know is that by calling IP 192.168.64.2:31068, you'll end up being routed to the WildFly Swarm pod on IP 172.17.0.2:8080 inside the cluster. But wait a minute—how do you know that it's port 31068 that ends up being routed to the WildFly Swarm application? To determine this, you can use kubectl:

```
$ kubectl get service
NAME           CLUSTER-IP     EXTERNAL-IP      PORT(S)
AGE
hello-service   10.0.0.204    <nodes>          8080:31068/TCP
3h
```

You can then see the port number (highlighted here). But there's an easier way: you can ask Minikube to give you the URL to any service running inside the cluster. To get the URL to hello-service, you type this:

```
$ minikube service --url hello-service
http://192.168.64.2:31068
```

If you run the command without the --url parameter, the URL opens automatically in the browser. And you get this response:

```
Swarm says hello from hello-world-1096927984-90q0p
```

KUBERNETES MANIFEST FILE

The last piece we want to show you about running applications in Kubernetes is the manifest file. At the beginning of section 18.3, we mentioned that you need a Docker image and a Kubernetes manifest file to deploy an application. So far, you've deployed applications via the kubectl command, which is capable of *autogenerating* the manifest file from the information given.

At any given time, you can use the kubectl tool to output any of the resources in the cluster as a manifest file. If you run the following command, it outputs in YAML format what the Kubernetes manifest file would have to be if you were to type that by hand. The output is 58 lines, and here are the first 10 lines of the manifest file:

```
$ kubectl get deployment -o yaml hello-world
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: 2017-11-04T20:33:36Z
  generation: 1
  labels:
    foo: bar
  name: hello-world
```

If you were to write this by hand, you wouldn't need all 58 lines; some could be skipped. But you'd still end up having to write 20 or more lines, which is tedious. Therefore, let's go back to being Java developers and use some nice tooling that can do all of this for us.

Running any Docker image in Kubernetes

The kubectl run command can be used to run any Docker image as a container in the Kubernetes cluster. DockerHub

is an online Docker image repository that hosts a lot of images. A beginner example is to run an Nginx container, which can be done as follows:

```
kubectl run my-nginx --image=nginx --port=80
```

Then you need to create a service to expose a network connection:

```
kubectl expose deployment my-nginx --type=NodePort --name=my-nginx-service
```

This allows you to call the service from your host operating system:

```
minikube service --url my-nginx-service  
http://192.168.64.2:30036
```

And you can then open a web browser and enter the URL:
<http://192.168.64.2:30036>.

In the next section, we want to start all over again, so let's delete what we created. First, delete the deployment:

```
kubectl delete deployment hello-world
```

Then delete the service as well:

```
kubectl delete service hello-service
```

That's it. You're now ready to start again and run Java applications using Maven.

18.3.3 RUNNING JAVA APPLICATIONS IN KUBERNETES USING MAVEN TOOLING

Any Java project that's Maven-based can quickly be turned into being Kubernetes-ready in less time than it takes to brew a cup of coffee or tea.

You turned the Camel microservices in section 18.2 into Docker images with the help of the Docker Maven plugin. You

could do the same thing now, but with the Kubernetes-ready fabric8 Maven plugin. You can quickly add this plugin to any Maven project by running the following command from the project directory:

```
cd chapter18/kubernetes/hello-swarm  
mvn io.fabric8:fabric8-maven-plugin:3.5.33:setup
```

Here you run the setup goal of fabric8-maven-plugin. But pay attention to how you can refer to the plugin using full Maven coordinates. This is a Maven standard that allows you to run any Maven plugin if you specify the full Maven coordinates (groupId:artifactId:version:goal).

Running the setup goal will add fabric8-maven-plugin to the Maven pom.xml file, and from this point onward you can run the plugin using just this:

```
mvn fabric8:nameOfGoal
```

Now you can easily deploy this application into the running Kubernetes cluster. Before doing so, remember to set up the command shell to point to the Kubernetes master:

```
eval $(minikube docker-env)
```

And then you can deploy the application:

```
mvn fabric8:deploy
```

TIP You can use `mvn fabric8:undeploy` to uninstall a deployed application. You can also undeploy an application by deleting the deployment from the `kubectl` command line:
`kubectl delete deployment nameOfDeployment.`

What happens now is that the plugin will scan your Maven project and then make a best effort to generate Kubernetes manifest files that fit the profile of your project. The plugin logs to the console what it has detected and what it's doing:

```
[INFO] F8: Building Docker image in Kubernetes mode  
[INFO] F8: Running generator wildfly-swarm  
[INFO] F8: wildfly-swarm: Using Docker image fabric8/java-jboss-openjdk8  
-jdk:1.2 as base / builder
```

Here you can see it's detected that WildFly Swarm is being used, and will then adapt accordingly. You can then use the `kubectl` tool to see the status of the pods:

```
$ kubectl get pods  
NAME                                     READY   STATUS  
RESTARTS      AGE  
helloswarm-kubernetes-2173069017-fpn7c   1/1     Running  
0           5m
```

You can see the pod is running, but you'd like to be sure and look inside the logs of the WildFly Swarm application. This can be done using `kubectl`:

```
kubectl logs helloswarm-kubernetes-2173069017-fpn7c
```

And you can use the `-f` option to follow the logs.

Another way to show the logs is to use `fabric8-maven-plugin`, as shown here:

```
mvn fabric8:log
```

This then dumps and follows the log of the running pod. To stop, press `Ctrl-C`.

Okay, so the WildFly Swarm application is running in the cluster. Now you want to call its service from your web browser. Yes, this is the same situation we illustrated previously in [figure 18.3](#). What you need to do is configure the service to be a `NodePort` type.

NOTE Using the `NodePort` type works well for local development on Minikube. But in a real cluster, you should use the `LoadBalancer` type, which will let Kubernetes use any built-in load balancer from the underlying infrastructure or cloud

provider.

This can be done by adding configuration to fabric8-maven-plugin in the pom.xml file. But there's an easier way. First, run this Maven goal:

```
mvn fabric8:resource
```

The plugin generates Kubernetes manifest files in the directory target/classes/META-INF/fabric8. You can then open the file kubernetes.yml and see all the glory of the Kubernetes manifest file. What you need to do is somehow merge in the details you want. The easiest way is to create a new file in your Maven project named src/main/fabric8/service.yml. The content of this file will then be merged into the service part of the Kubernetes manifest file. To enable NodePort, you add the following lines:

```
spec:  
  type: NodePort
```

And then you can run `mvn fabric8:resource` to regenerate the Kubernetes manifest file. In target/classes/META-INF/fabric8, you can see whether your changes have been included.

CONFIGURING NODEPORT IN POM.XML

Instead of configuring the Kubernetes manifest in the src/main/fabric8/service.yml file, you can do most of the configuration from fabric8-maven-plugin in the pom.xml file. You may want to do this when you set only a smaller number of configurations, such as setting the port type to be NodePort. But this requires a fair bit of XML, as you need to add the following lines:

```
<configuration>  
  <enricher>  
    <config>  
      <fmp-service>  
        <type>NodePort</type>  
      </fmp-service>
```

```
</config>
</enricher>
</configuration>
```

What's best to use? Using the YAML file, you configure the Kubernetes manifest directly, whereas using pom.xml, you're using an XML configuration that resembles the Kubernetes manifest structure but isn't exactly the same. At the end of the day, it's good to familiarize yourself with the Kubernetes manifest structure, so using the YAML files is likely better.

CALLING THE SERVICE

To try calling the service, you need to get the service URL from Minikube:

```
$ minikube service --url helloswarm-kubernetes
http://192.168.64.2:30102
```

And then type the URL in the web browser:

```
http://192.168.64.2:30102
```

This returns a response from the running WildFly Swarm application:

```
Swarm says hello from helloswarm-kubernetes-2176083623-
fshwg
```

You can also let Minikube open your web browser and call the service if you omit the --url parameter:

```
minikube service helloswarm-kubernetes
```

Now let's build and run the Spring Boot client that calls this service. This should be easy, right? It is—but there's always something lurking.

18.3.4 JAVA MICROSERVICES CALLING EACH OTHER IN THE CLUSTER

What you want to do now is deploy and run the Spring Boot client microservice that should call the existing WildFly Swarm

microservice. One microservice will call the other microservice, with both of them running in the Kubernetes cluster. As you may remember from section 18.1.4, when you were using Docker containers to call each other, you had to change the hostname from localhost to the IP address of the WildFly Swarm container. In Kubernetes, you use Kubernetes services when you want to connect to services—for example, other applications—over the network. [Figure 18.4](#) illustrates this principle.

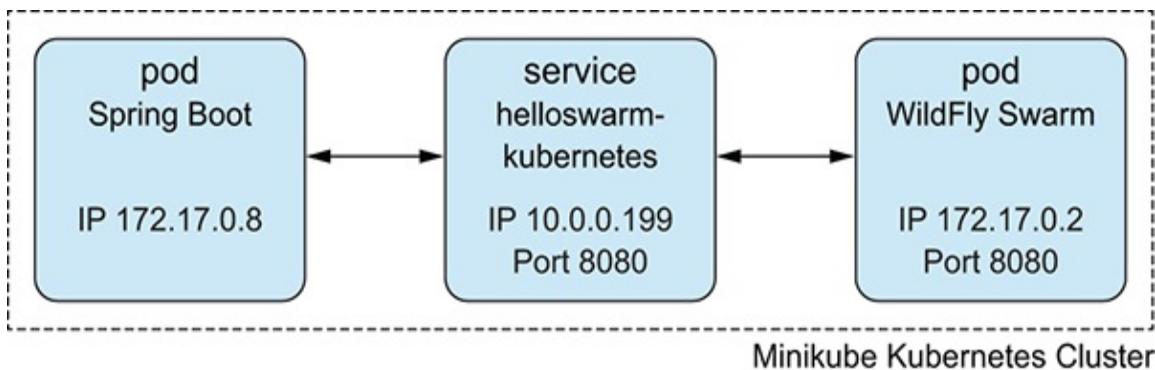


Figure 18.4 The Spring Boot microservice wants to call the Hello service from the WildFly Swarm microservice. In Kubernetes, you use services to accommodate this. The service sits between the clients and the pods hosting the service and acts as local gateway and load balancer to direct traffic to running pods that are ready to service the traffic.

What does this mean for us as Java developers? It means that every time we want to call a microservice in the cluster, we have to use a Kubernetes service. At this time, you can think of a Kubernetes service as a level of indirection that sits between you and the service. In [figure 18.4](#), you can see the service in the middle.

Camel makes it easy to call a Kubernetes service. Here's the Camel route that would call the service:

```
from("timer:foo?period=2000")
    .to("netty4-http://localhost:8080")
    .log("${body}");
```

You need to change this to use the IP address and port of the service:

```
from("timer:foo?period=2000")
```

```
.to("netty4-http://10.0.0.199:8080")
.log("${body}");
```

But how do you know the IP address? As with Docker, you can use command-line tooling to retrieve such details:

```
$ kubectl describe service helloswarm-kubernetes
Name:           helloswarm-kubernetes
Type:          NodePort
IP:            10.0.0.199
Port:          http    8080/TCP
NodePort:      http    30102/TCP
```

Here you can see that the IP address is 10.0.199 and the port number is 8080. After this code change, you could build and run this application in the cluster as follows:

```
cd chapter18/kubernetes/client-spring
mvn fabric8:deploy
```

But now you hardcode an IP address and port number in the source code, and isn't this bad? Well, yes and no. The brilliance of a Kubernetes service is that it has a static IP address and port number for the entire lifecycle of the service. On the other hand, pods are temporary and can come and go, and each time the pod may have a different IP address.

But let's get rid of the IP address, as Camel makes this easy. You can refer to a Kubernetes service in Camel using the {{service:name}} syntax. The route can be changed to the following:

```
from("timer:foo?period=2000")
.to("netty4-http://{{service:helloswarm-kubernetes}}")
.log("${body}");
```

With Camel, you have another choice instead of using {{service:name}}: you can use the Service Call EIP and write the route as follows:

```
from("timer:foo?period=2000")
.serviceCall("helloswarm-kubernetes")
.log("${body}");
```

The Service Call EIP works the same as `{{service:name}}`, but the EIP makes it stand out in the route with a strong indication that it's calling a service. The book's source code contains an example of using the Service Call EIP, which you can find in the `chapter18/kubernetes/client-spring-servicecall` directory. The preferred Kubernetes practice is to use DNS. The preceding example can be written as follows:

```
from("timer:foo?period=2000")
    .to("netty4-http://helloswarm-kubernetes:8080")
    .log("${body}");
```

When using the DNS name, you must remember to use the port number as well. Because you're not using the default HTTP port number 80, you have to specify the port number 8080. In the earlier versions of Kubernetes, DNS wasn't supported, so your only choice was to use `{{service:name}}` or the Service Call EIP. At the time of this writing, DNS is so well supported that it's the recommended choice.

Okay, now we're happy, so let's run the client.

TIP Section 18.4 goes into more depth about Kubernetes services as well as other Kubernetes concepts.

But let's show you another feature of `fabric8-maven-plugin` that we like. You can use the `run` goal to make it appear that you're running your current Java project locally in the foreground:

```
cd chapter18/kubernetes/client-spring
mvn fabric8:run
```

What happens now is the same as `fabric8:deploy`, but the plugin will appear to run locally by running in the foreground and following the log. From the logs, you should see that the client is able to call the service:

```
Swarm says hello from helloswarm-kubernetes-171471280-d8r0k
Swarm says hello from helloswarm-kubernetes-171471280-d8r0k
```

Swarm says hello from helloswarm-kubernetes-171471280-d8r0k

Press Ctrl-C to stop the plugin, which undeploys the application.

Open the Kubernetes web console:

```
$ minikube dashboard
```

Then you can see your running pods and more information. Spend a couple minutes browsing the console, shown in figure 18.5.

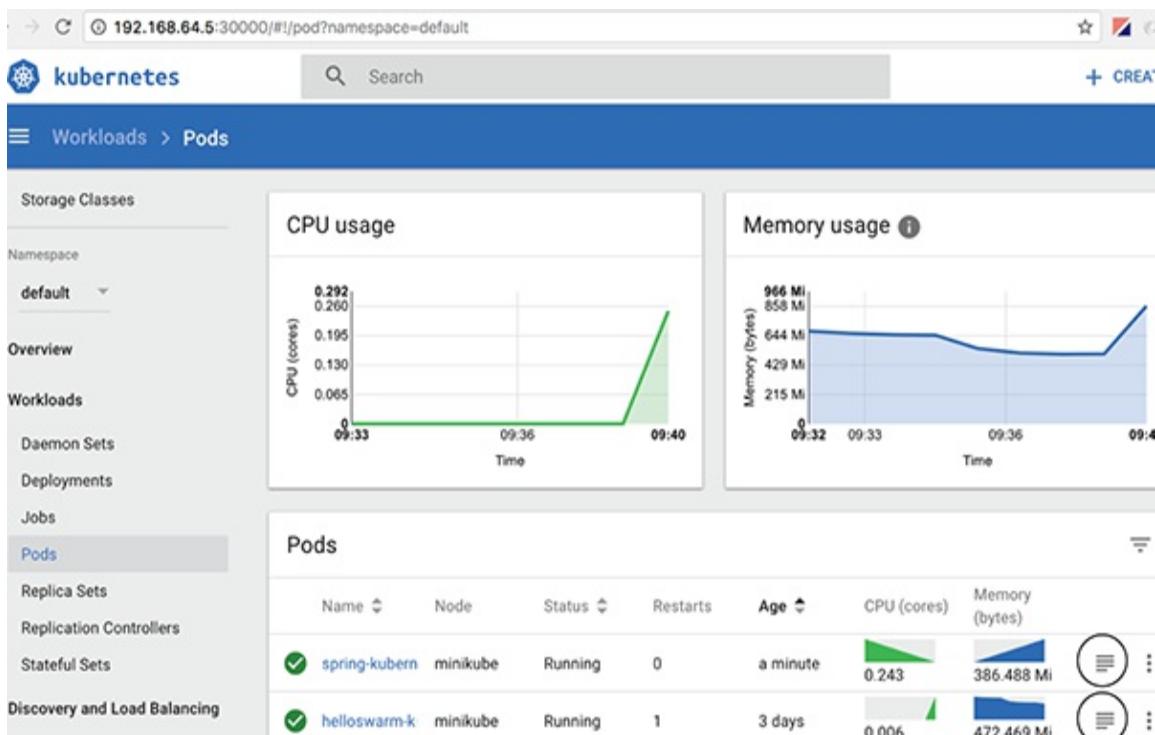


Figure 18.5 The Kubernetes web console shows the running pods. From the console, you can access the logs of the pods by clicking the second-to-last button (marked with a circle) on the right side. The two diagrams at the top show the cluster-wide CPU and memory usages. Likewise, you can see the CPU and memory usage for each pod as well in the tiny graphs shown in the table.

TIP To display CPU and memory usage graphs in the dashboard, you must first install the heapster add-on in Minikube.

As a Java developer, you might worry that having to run your

Java applications in the Kubernetes environment means your life will become more problematic. For example, traditional Java applications can easily be started or debugged directly from the Java editor. Is that even possible with Kubernetes, and if so, how can you do that?

18.3.5 DEBUGGING JAVA APPLICATIONS IN KUBERNETES

You can debug your Java applications running in the Kubernetes cluster, even if the cluster isn't running locally with Minikube. Imagine that you suspect that the WildFly Swarm microservice has a bug, and you want to debug the microservice while it runs in the Kubernetes cluster.

You can do this with help from the fabric8 Maven plugin, using its debug goal. But before you can use this, you must have an existing running pod.

DEBUGGING A RUNNING POD

If you haven't deployed the WildFly Swarm application, you can do so with the deploy goal:

```
cd chapter18/kubernetes/hello-swarm  
mvn fabric8:deploy
```

Now imagine that when you call the Hello service, it returns a response that we suspect is caused by a bug:

```
$ curl http://192.168.64.2:30102  
Swarm says hello from helloswarm-kubernetes-3397218272-  
pptf4
```

You now want to debug the running pod with the WildFly Swarm microservice. This is done by running the debug goal of the fabric8 Maven plugin:

```
cd chapter18/kubernetes/hello-swarm  
mvn fabric8:debug
```

What happens next is that the running pod is enabled in debug

mode, which may take a little while. The plugin logs in the command shell what's happening:

```
[INFO] F8: Enabling debug on Deployment helloswarm-kubernetes
[INFO] F8: Waiting for debug pod with selector
  LabelSelector(matchExpressions=[], matchLabels=
{project=helloswarm
-kubernetes, provider=fabric8, version=2.0-SNAPSHOT,
group=com.camelinaction}, additionalProperties={}) and
$JAVA_ENABLE_DEBUG
  = true
[INFO] F8:[W] helloswarm-kubernetes-3781014250-cs2f1
status: Pending
[INFO] F8:[W] helloswarm-kubernetes-3397218272-pptf4
status: Running Ready
[INFO] F8:[W] helloswarm-kubernetes-3781014250-cs2f1
status: Pending
[INFO] F8:[W] helloswarm-kubernetes-3781014250-cs2f1
status: Running Ready
[INFO] F8: Port forwarding to port 5005 on pod helloswarm-
kubernetes
-3781014250-cs2f1 using command /usr/local/bin/kubectl
[INFO] F8:
[INFO] F8: Now you can start a Remote debug execution in
your IDE using
  localhost and the debug port 5005
[INFO] F8:
[INFO] F8:kubectl Forwarding from 127.0.0.1:5005 -> 5005
[INFO] F8:kubectl Forwarding from [::1]:5005 -> 5005
```

The pod has been restarted with the `JAVA_ENABLE_DEBUG` environment variable set to `true`. All the Java base images from fabric8 or JBoss support Java debugging controlled by this environment variable. They start Java with remote debugging enabled on port `5005`. To make it possible to remotely debug from your host operating system, port forwarding from Kubernetes has been enabled on port `5005` as well. In other words, you can do a Java remote debug on `localhost:5005`, and it'll be network connected to the Kubernetes cluster and the running pod. From your Java editor such as IDEA, you can then do a remote Java debug on port `5005` and set a breakpoint in the source code, as shown in [figure 18.6](#).

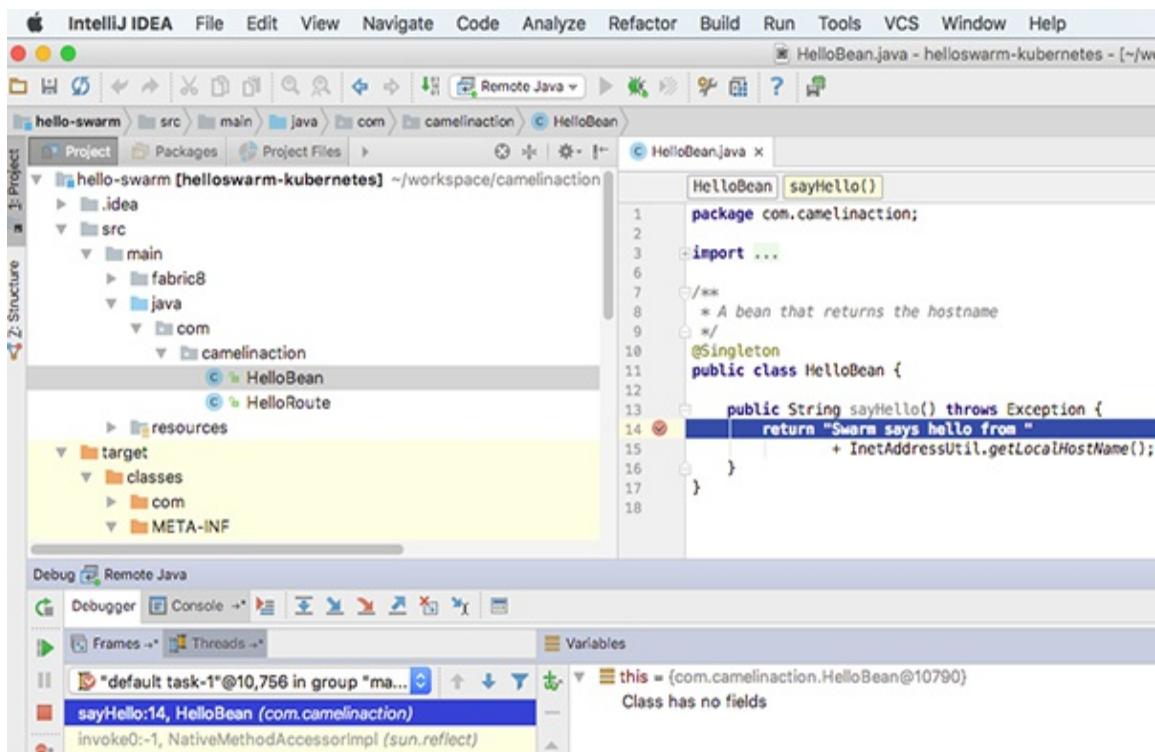


Figure 18.6 Remote debugging Java from an IDEA editor. The Java application runs in a pod in the Kubernetes cluster, which you've network connected using the fabric8-maven-plugin:debug goal to make this easy.

From the Java editor, you can then inspect what's happening, and even change the response message. For example:

```
$ curl http://192.168.64.2:30102
I changed this in the editor
```

This is awesome. You can call the service again, and the remote debugger is activated again. To stop the debugger, press Ctrl-C in the command shell where the fabric8-maven-plugin:debug is running.

TIP To speed up the debugging, you can deploy the pod in debug mode from the beginning:

```
mvn fabric8:deploy -Dfabric8.debug.enabled=true
```

CALLING A SERVICE IN KUBERNETES KEEPS CHANGING THE PORT NUMBER

When calling a service from your host operating system into the Kubernetes cluster, you have to expose the service using the NodePort type. The exposed service may be re-created when the pod is redeployed or debugged. This can be annoying, because when you attempt to call the service with the old port number, it will fail:

```
$ curl http://192.168.64.2:30102  
curl: (7) Failed to connect to 192.168.64.2 port 30102:  
Connection refused
```

Here's a trick for avoiding this. You can specify the fabric8 Maven plugin to avoid triggering a service change in Kubernetes by setting an ignore option when debugging:

```
mvn fabric8:debug -Dfabric8.deploy.ignoreServices=true
```

It also works for re-deploying as well:

```
mvn fabric8:deploy -Dfabric8.deploy.ignoreServices=true
```

You've now climbed to the top of the Kubernetes mountain, and that's enough action for today. Let's take a moment to reflect on what you've achieved.

RECAP

At the beginning of this chapter, you set out on a course to build two small Java (with Camel) microservices and make them deploy in a local Kubernetes cluster, all running on your local computer. You did this in three steps. First, you built the microservices and ran them standalone. Then you moved on to add Docker into the mix; you built the microservices as Docker images, which you could run as Docker containers. Then Kubernetes entered the stage, and you first had to install a local Kubernetes cluster called Minikube.

You learned a few tips and tricks for using the Kubernetes command-line tool (kubectl) to deploy and manage applications in the cluster. Then you returned to your Java microservices, and with the help from the fabric8-maven-plugin, you could easily

build and run your microservices in the cluster. You also learned how to call from your host operating system into Kubernetes, to call services running inside the cluster. Java applications running in the cluster can also be debugged easily with the fabric8:debug goal, which you learned how to use.

Speaking of Kubernetes, let's go over the most essential concepts it brings to the table. You are now half way through this chapter, and it may be a good time for a little break.

18.4 Understanding Kubernetes

We've already had our fun with Kubernetes in the previous section, where we were running a single-node cluster using Minikube. We hope that with some practical experience, you're now better suited for what's coming in this section. First, we'll explain some of the reasons that container-managed platforms such as Kubernetes are becoming increasingly needed by businesses.

To become skilled with Kubernetes, you'll go over its essential concepts and architecture. With this information, you can then jump back into action in the following sections to scale containers and provoke failures and learn how to make Camel microservices truly cloud native.

18.4.1 INTRODUCING KUBERNETES

Today we're on the verge of changing the way we build, package, and run our applications. Traditionally, we've built monolithic applications that run on application servers, together with other applications within the same JVM (OS process). We're heading toward breaking up these big monoliths into smaller pieces known today as *microservices*. Because these microservices are decoupled from each other, we can develop, build, deploy, run, and scale them individually. From the business side of things, we'll be able to deliver business values faster in order to keep up with today's rapidly changing business requirements.

But with a growing number of applications to be deployed, it becomes increasingly more difficult to configure, manage, and keep the infrastructure up and running. Businesses are also always on the lookout for ways to reduce costs, and one way is to better use resources in their data centers. Managing all this complexity isn't sustainable with manual procedures and labor. We need a high degree of automation and an infrastructure that's self-managed and capable of orchestrating, scheduling, configuring, handling failures in, and supervising the platform as a whole. The answer to that is Kubernetes.

Kubernetes enables developers to deploy applications themselves without requiring assistance from system administrators. The lives of system administrators are also improved as they shift from focusing on managing individual applications to instead managing Kubernetes and the rest of the infrastructure.

Kubernetes abstracts away all the hardware complexity and exposes your data centers as a unified computing resource platform. It allows you to deploy and run applications without having to worry about the hardware and servers underneath. When you deploy applications on Kubernetes, you let Kubernetes orchestrate and schedule which servers each individual component of the applications is running on. Communication between services and applications in the cluster is made easy by Kubernetes, which makes it an ideal platform for distributed applications. Kubernetes can easily scale applications up and down as needed without affecting the network communication between applications and services. For example, Kubernetes makes it possible to move workloads between nodes without affecting applications or downtime of services.

Kubernetes does all that and does it so well that it's becoming the de facto standard for running distributed applications in both the cloud and on on-premises infrastructure.

WHERE DID KUBERNETES COME FROM?

The short answer is Google. Google has been running containers at scale in its data centers for more than 10 years. And Google has become so good at doing this that everything in Google runs in containers. Google has built several container-management platforms over time, and Kubernetes is its last and best effort. It's based on all of Google's knowledge and experience.

In 2014, Google decided to open source its next-generation successor to Borg (its current platform), named Kubernetes. Since then, Kubernetes has become a large, open, and vibrant community with contributors from many IT vendors such as Google, Microsoft, Intel, IBM, CoreOS, Red Hat, and many more. To ensure an even playing field, Google transferred Kubernetes to the Cloud Native Computing Foundation, where the project is governed by a board comprising various stakeholders and companies. The success of Kubernetes has skyrocketed, and it's also popular in the developer community, where it's the number one project on GitHub.

Are you ready to learn about the high-level architecture of Kubernetes? Read on.

18.4.2 KUBERNETES ARCHITECTURE

A high-level, 10,000-feet diagram of Kubernetes is sketched in figure 18.7.

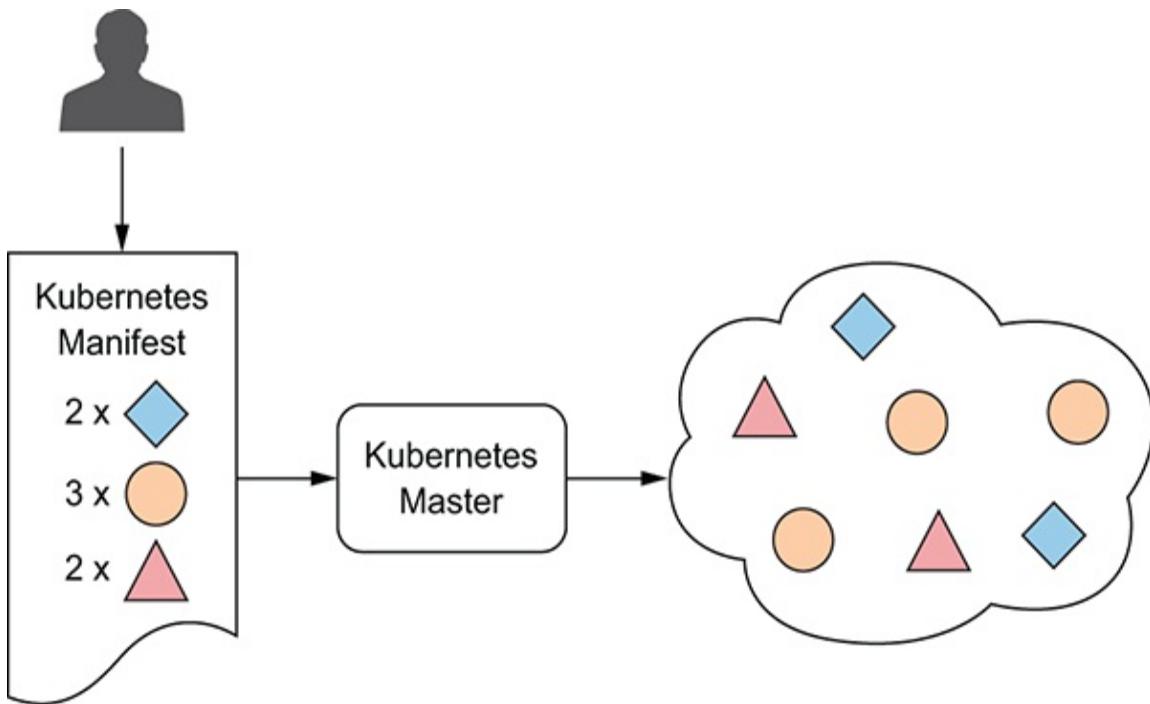


Figure 18.7 The cloud shape represents the worker nodes that Kubernetes exposes as a single container-based platform. The Kubernetes master orchestrates and supervises the platform. Users can deploy applications by applying Kubernetes manifest files to the Kubernetes master.

The Kubernetes platform includes a master node (in reality you would run multiple instances of the master node to make the cluster highly available) that orchestrates numerous work nodes (represented as the cloud). When a developer submits a list of applications (in a Kubernetes manifest file) to deploy to the master, Kubernetes then deploys these applications in the cluster of worker nodes. It doesn't matter which worker node an application is deployed on. It's all decided by Kubernetes, which has built-in intelligence to orchestrate the platform in the best possible way.

DECLARATIVE STATE

After the applications are running, Kubernetes will continuously make sure that the platform runs the desired number of applications, as described in the Kubernetes manifest files. For example, in figure 18.7, three applications are set to run two, three, and two instances, respectively. If one of those instances

stops for any reason (for example, the process crashes or becomes unresponsive), Kubernetes will restart it automatically. Likewise, if a worker node dies or becomes inaccessible, Kubernetes will reschedule the applications to run on the remaining working nodes.

This notion of declarative state is a key principle of Kubernetes. We as users should just specify what the desired state should be and leave it up to Kubernetes to ensure that the cluster runs accordingly.

When working with Kubernetes, you should familiarize yourself with the concepts that follow.

18.4.3 ESSENTIAL KUBERNETES CONCEPTS

This section covers essential Kubernetes concepts that we think are the most beneficial:

- Pods
- Labels
- Replication controllers and deployments
- Services

PODS

A *pod* is a group of one or more Docker containers (like a pod of whales). A typical pod has only one Docker container. When you need two or more containers (known as the *side-car* or *ambassador* pattern), a pod is the way to group them together.

A pod is the minimal deployment unit in Kubernetes, and pods are orchestrated, scheduled, and managed by Kubernetes. When we refer to an application running in Kubernetes, it's running inside a pod as a Docker container. A pod is given its own IP address, and all containers within the pod share this IP as well as the same local network and filesystem. Likewise, if persistent volumes are mounted to a pod, they're also shared among the containers running in the pod.

An important detail to know about pods is that they’re ephemeral (temporary). Pods can come and go in the cluster—for example, pods can be scaled down, or containers within the pod may crash. An application can run in a pod with a given IP address, and then because of a crash, the application can be started on another pod with a new IP address. This falls in line with the world of microservices and distributed systems, where things will fail, so you’re strongly encouraged to write your applications with this concept in mind.

LABELS

So far in this chapter, you’ve been running a few pods in your local Minikube cluster. When running Kubernetes for real, you’ll end up with many more pods running in the cluster. You need a way of being able to organize and group these pods, and the answer to this is labels.

Labels are just key-value pairs that can be assigned to pods—for example, `tier=frontend` or `tier=backend`. A pod can also have multiple labels, such as `tier=backend, app=inventory`.

After labeling your pods, you’ll be able to query the cluster using label selectors to find the pods belonging to a certain group. For example, to find all front-end pods, you can use a label selector with `tier=frontend`. You can also find all pods that aren’t front end using `tier!=frontend`. As a Camel fan, you can also add a label to your pods to indicate that the pods run Camel.

You can use the `--show-labels` parameter to show all the labels for the pods:

```
$ kubectl get pods --show-labels
```

As an example, suppose you have the label `animal`; you can then list all pods running Camel using `kubectl`:

```
$ kubectl get pods -l animal=camel
```

NAME	READY	STATUS
RESTARTS AGE		
helloswarm-kubernetes-171471280-d8r0k	1/1	Running 1

Kubernetes has no restrictions or standards on the key names of the labels (hence our use of `animal` in the preceding example).

You can use any arbitrary name you desire. From the Kubernetes point of view, it doesn't matter whether the keys are named `tier`, `version`, `animal`, `bambi`, or `thunderstruck`.

REPLICATION CONTROLLERS AND DEPLOYMENTS

As you've learned, pods represent the deployment unit in Kubernetes. You can create, manage, and supervise your pods manually, but in real life you want your pods to stay up and run automatically without manual intervention. Instead of creating pods manually, you'd create other types of resources, such as replication controllers or deployments, to then create and supervise the pods.

A *replication controller* is a Kubernetes resource that ensures a pod is always up and running. If the pod dies or become unresponsive, the replication controller creates a new pod immediately.

The concept of replication controllers comes back to the notion of desired state in Kubernetes. The replication controllers constantly monitor the running pods and ensure that the number of pods always matches the desired state. If not enough pods are running, the controller creates new pods based on a pod template. If there are too many pods, excess pods are stopped and removed.

A replication controller has these three parts:

- *Label selector*—Groups pods controlled by the replication controller
- *Replica count*—Specifies the desired number of running instances
- *Pod template*—Describes how to create a new pod

In section 18.3, you deployed our two Camel microservices in the cluster; both had been set to a replicate count of 1 by default. You could use the replication controller to scale up the number of pods—for example, to run two instances of the WildFly Swarm application. This can be done using the kubectl command-line tool:

```
$ kubectl scale --replicas=2 deployment/helloswarm-kubernetes  
deployment "helloswarm-kubernetes" scaled
```

The list of pods now shows two pods with the WildFly Swarm application:

```
$ kubectl get pods  
NAME                                     READY   STATUS  
RESTARTS   AGE  
helloswarm-kubernetes-171471280-d8r0k    1/1     Running   1  
1d  
helloswarm-kubernetes-171471280-nlpw1     1/1     Running   0  
20s
```

You may have noticed that you specified deployment in the arguments to the kubectl command when you scaled the WildFly Swarm application up to two replicas. This is because you’re using Kubernetes deployment to deploy applications, which was a concept introduced in a later release. A Kubernetes deployment is a higher-level replication controller that’s capable of performing more-advanced rollout and rollback of pods. You can use the kubectl tool to list the current deployments:

```
$ kubectl get deployment  
NAME          DESIRED   CURRENT   UP-TO-DATE  
AVAILABLE   AGE  
helloswarm-kubernetes   2         2         2         2  
1d  
spring-kubernetes       0         0         0         0  
1d
```

Here you can see that we desired two pods of the WildFly Swarm application, which are also the current number of instances running. On the other hand, the Spring Boot application was scaled down to zero, which means no pods are running currently.

This is a common way to stop an application—by scaling it down to zero.

TIP You can use the `describe` command from `kubectl` to output verbose details about any Kubernetes resource. For example: `kubectl describe deployment helloswarm-kubernetes`.

SERVICES

Kubernetes services are brilliant. They’re as simple as possible, but elegant in their solution. Why? First, let’s set the scene. You’ve learned about pods, and replication controllers ensure that a certain number of pods are constantly running and working as intended. In a microservice or distributed architecture, your applications often need to communicate with each other, and outside the cluster as well—for example, to listen to HTTP requests coming from clients outside the cluster, to call into services hosted on these pods. These pods need a way of discovering each other so they can communicate.

You also learned that Kubernetes can scale up pods so that multiple instances run in the cluster. Therefore, when you try to communicate with pods, you shouldn’t rely on their direct IP addresses because pods can come and go (they’re dynamic). What you need is a level of indirection, where you can group the pods that provide the service, the way your applications communicate with them, and possibly load-balance against them.

This is exactly what a Kubernetes service is. It allows you to use a label selector to group your pods, and abstract them with a single static network IP address that you can use to discover and communicate with. A Kubernetes service is a single IP address for a group of pods that provide the same service. The IP address is static and never changes while the service exists. You previously encountered a Kubernetes service, illustrated in [figure](#).

18.3. This time, let's use the two Camel microservices with Spring Boot and WildFly Swarm as an example. In [figure 18.8](#), we've emphasized where the Kubernetes service lives and how it sits between the Spring Boot and WildFly Swarm pods.

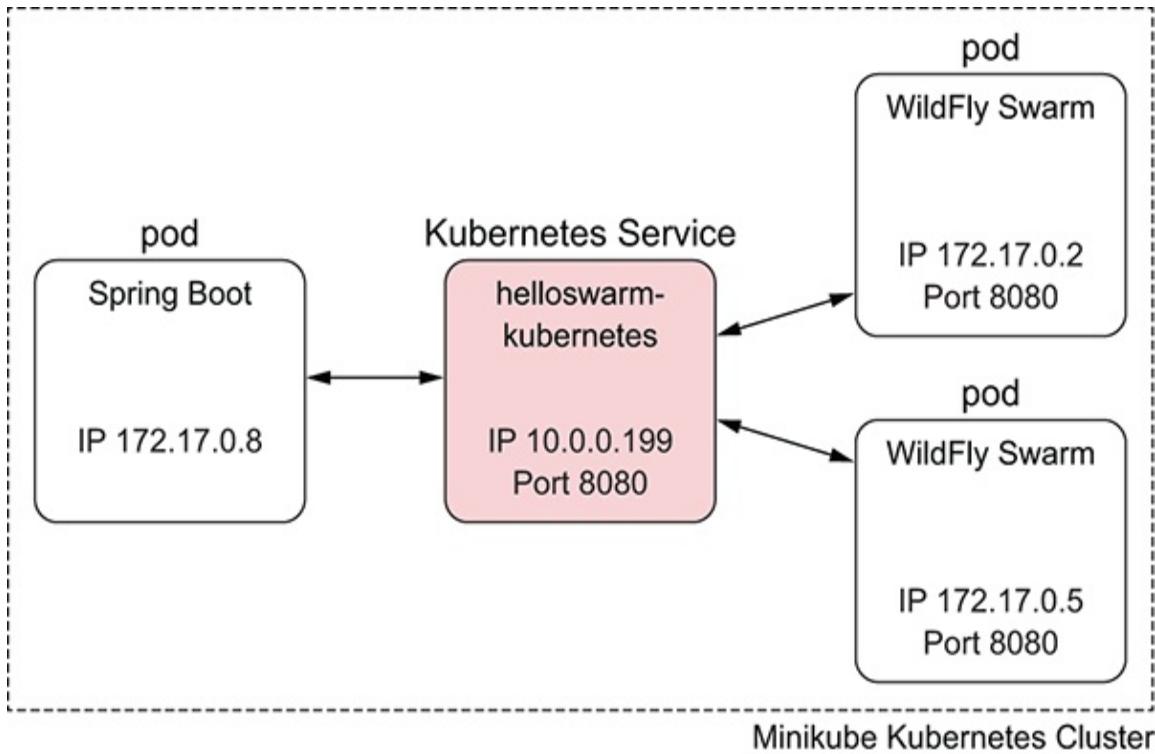


Figure 18.8 The Spring Boot pod calls the helloswarm-kubernetes service, which by itself is a static virtual network IP on 10.0.0.199 that traps the traffic and then load-balances between the running pods that provides the service.

A Kubernetes service consists of the following parts:

- Unique name
- Label selector
- Static IP address and port number

For example, you can use the `kubectl` tool to view details of services:

```
$ kubectl describe service helloswarm-kubernetes
Name:           helloswarm-kubernetes
Namespace:      default
Selector:      group=com.camelinaction,
               project=helloswarm-
```

```
kubernetes,provider=fabric8
Type:           NodePort
IP:             10.0.0.199
Port:            http      8080/TCP
NodePort:        http      30102/TCP
Endpoints:      172.17.0.2:8080,172.17.0.5:8080
Session Affinity: None
```

Here you can see that the service has the static IP of 10.0.0.199 and uses port 8080. You can also see that currently the two pods that provide the service are listed with their IP addresses under Endpoints. The Selector refers the label selector that's used to gather the list of pods. You can run a `kubectl get pods` command with the selector to see the list of pods:

```
$ kubectl get pods -l
group=com.camelinaction,app=helloswarm-
kubernetes,provider=fabric8
NAME                           READY   STATUS
RESTARTS   AGE
helloswarm-kubernetes-171471280-d8r0k   1/1     Running   1
1d
helloswarm-kubernetes-171471280-nlpw1   1/1     Running   0
2h
```

As expected, the two WildFly Swarm pods provide the service.

Let's review one more time why Kubernetes services are brilliant. They're brilliant because they're an elegant abstraction. From a client point of view, a service is just a static IP address and port number that never changes. There's no need for any dynamic lookup to a service registry to obtain a list of locations where the services are currently running. It can't be simpler for clients to call a service: just a regular network call on a static IP address and port number.

And the brilliance doesn't stop there. Kubernetes also provides a service call using DNS. Instead of calling the service using its static IP and port number, you can call using its service name.

For example, you can call the WildFly Swarm service from within the Kubernetes cluster. To do this, you can start bash on the running Spring Boot pod:

```
kubectl exec -it spring-kubernetes-3845155382-ww31q bash
```

From the bash shell, you can use curl to call the service, at first using the direct IP addresses of the pods:

```
$ curl http://172.17.0.2:8080
Swarm says hello from helloswarm-kubernetes-171471280-d8r0k
$ curl http://172.17.0.5:8080
Swarm says hello from helloswarm-kubernetes-171471280-nlpw1
```

Then you can call the IP address of the service itself:

```
$ curl http://10.0.0.199:8080
Swarm says hello from helloswarm-kubernetes-171471280-nlpw1
```

And you can also use the DNS name to call the service. Notice how it load-balances among the two running pods, by returning a different hostname in the response:

```
$ curl http://helloswarm-kubernetes:8080
Swarm says hello from helloswarm-kubernetes-171471280-nlpw1
$ curl http://helloswarm-kubernetes:8080
Swarm says hello from helloswarm-kubernetes-171471280-d8r0k
```

Using the kubectl exec command is a handy trick to connect to a running pod and poke around to see what's happening from the bash shell.

OTHER CONCEPTS

After you have more experience using Kubernetes, you should become familiar with other concepts that are good to know. Here are a few:

- *Namespace*—A virtual cluster within the cluster. You use namespaces to separate clusters.
- *Probes*—Used for probing when pods are ready to service traffic and whether they're alive. Section 18.5 covers this topic.
- *PetSet*—Used for stateful applications.
- *Jobs*—Used for tasks that run to completion and terminate.

- *Ingress*—Used to expose services to the outside using HTTP/HTTPS routes.
- *Persistent volume claims*—Used to mount persistent volumes to pods.
- *ConfigMap*—Can be used for configuration management.
- *Secrets*—Used to store sensitive information.

TIP If you want to learn all about Kubernetes, we suggest *Kubernetes in Action* by Marko Luksa (Manning, 2017).

It's time to get back on the Camel and into action. Let's have some fun and see what happens when we release the chaos monkey and start killing pods. How resilient and well behaved do you think our two small Camel microservices are in the cluster?

18.5 Building resilient Camel microservices on Kubernetes

Chapter 7's section 7.4 covered how to design for failures in a microservice architecture. This doctrine is of the utmost importance in any distributed system. In this section, we'll play devil's advocate and cause havoc in our Kubernetes cluster so you can witness what happens when your Camel microservices are not fault-tolerant.

We'll then discuss which Kubernetes features can be used to aid developers building fault-tolerant microservices. Let's begin by setting up the scene before we start being naughty.

18.5.1 SCALING UP MICROSERVICES

We begin our journey with our two Camel microservices running in the Kubernetes cluster. There's a pod of each, as shown here:

\$ kubectl get pods		READY	STATUS
NAME	RESTARTS AGE		
helloswarm-kubernetes-171471280-88vgw	12h	1/1	Running 1
spring-kubernetes-2151443245-27s8g	1m	1/1	Running 0

In case you're wondering why the WildFly Swarm pod has been restarted one time, it's because at the time it was the only pod running in the cluster, and Minikube was stopped. When Minikube is started again, it restarts the previously running pods.

In this example, the Spring Boot pod is the client that will continuously call the service running on the WildFly Swarm pod. We'll exploit this fact and see what happens with the client when the WildFly Swarm pod is being killed, scaled up and down, and what else can happen.

But from the beginning, everything is happy, and we'll follow the logs from the command shell:

```
$ kubectl logs -f spring-kubernetes-2151443245-27s8g
...
Swarm says hello from helloswarm-kubernetes-171471280-88vgw
Swarm says hello from helloswarm-kubernetes-171471280-88vgw
```

DELETING THE POD

Let's first try to see what happens if we delete the WildFly Swarm pod:

```
$ kubectl delete pods helloswarm-kubernetes-171471280-88vgw
pod "helloswarm-kubernetes-171471280-88vgw" deleted
```

How does the client behave? Not too well, as you'll shortly see: the client fails and outputs exceptions in the log:

```
Caused by: io.netty.channel.ConnectTimeoutException:
connection timed out: /10.0.0.199:8080
```

But then after a little while, the client works again and logs the response from the WildFly Swarm pod:

```
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
```

Notice that the hostname in the response has changed. That's because the pod was deleted and Kubernetes created a new pod, which gets a new IP address assigned and a new hostname as well.

You can find the new IP address (the previous IP address was 172.17.0.2) by running the `describe` command on the pod:

```
$ kubectl describe pod helloswarm-kubernetes-171471280-b9bj5
IP: 172.17.0.6
```

Oh, wait a minute—you almost fooled us. You told Kubernetes to delete the pod. This isn't exactly the same as if the pod dies because the process crashes or something that isn't in the direct hands of Kubernetes. Yes, you're right in telling Kubernetes to delete a pod, as this allows Kubernetes to perform this in a more graceful manner. Let's try to be more evil and kill the pod using Docker.

KILLING THE POD

To use the Docker CLI, you need to prepare your command-line shell, which you do from Minikube:

```
minikube docker-env
```

Then you run the `eval` command:

```
eval $(minikube docker-env)
```

You can now use Docker to list all the running Docker containers in the Kubernetes cluster:

```
$ docker ps
CONTAINER ID      IMAGE
563728aead40      camelinaction/helloswarm-
kubernetes:snapshot-...
```

You should see a big list of containers, but at the top, you may find the Docker process that runs the WildFly Swarm pod:

Now you can kill that pod using Docker and its container ID:

```
docker kill 563728aead40
```

The Spring Boot client should start failing again, and you should see errors and stacktraces in the log. But after a while, the WildFly Swarm pod comes back online, and the log outputs successful responses again:

```
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
```

This time, the pod was killed, and Kubernetes was able to self-heal by restarting the pod. Wait a minute, did we say *restart*? Yes. As you'll notice, the hostname in the response is the same as before you killed the pod. Also if you check the IP address of the pod, you'll see it's still the same as before. But its restart count is now 1:

```
$ kubectl get pod
NAME                               READY   STATUS
RESTARTS   AGE
helloswarm-kubernetes-171471280-b9bj5   1/1    Running   1
24m
spring-kubernetes-2151443245-27s8g      1/1    Running   0
26m
```

Whether the pod is deleted or killed, Kubernetes is able to self-heal and either start a new pod or restart the failed Docker container in the existing pod. Either way, Kubernetes does what's necessary to keep the state of the running cluster according to the desired state. And what's the desired state? You can see this from the deployments:

```
$ kubectl get deployment
NAME          DESIRED   CURRENT   UP-TO-DATE
AVAILABLE     AGE
helloswarm-kubernetes   1         1         1         1
2d
spring-kubernetes       1         1         1         1
2d
```

This is what would be expected. We're not falling off our chairs

yet. If you have only one instance of a pod running that provides a service, and if that pod dies, then clients attempting to call that service will fail. You can make this more resilient by scaling up the number of instances that provide the service.

SCALING UP THE POD

You scale up the WildFly Swarm microservice by setting the replicas on its deployment via the kubectl tool:

```
kubectl scale deployment helloswarm-kubernetes --replicas=2
```

You'd then expect the client to load-balance between the two running pods. Oh, what just happened? Did you spot a stacktrace in the client log?

```
Cannot connect to 10.0.0.199:8080
```

But among those stacktraces are some positive responses:

```
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
```

And after a little while, the client gets only successful responses:

```
Swarm says hello from helloswarm-kubernetes-171471280-pdkwv
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
```

What just happened? The WildFly Swarm deployment was scaled up from one to two pods, which means Kubernetes created and started a new pod. The client that continuously calls the service will now start to load-balance between both pods. That is why you can see two hostnames in the preceding logs. The load balancing in Kubernetes is by default random, so that's why you see two consecutive responses from the same pod in the last two lines.

But what about those errors and stacktraces you saw in the beginning? Those occur because when Kubernetes starts up the second pod, that pod is included in the service load balancing right away and therefore receives traffic from the calling client. But the pod isn't yet ready. It's a Java application and it takes a

while to start up and initialize WildFly Swarm and Camel, and only then is it ready to receive calls.

To fix this problem, Kubernetes has a couple of health probes that should be used.

18.5.2 USING READINESS AND LIVENESS PROBES

Kubernetes allows you to define a readiness probe on your pod's containers. The readiness probe will be called periodically to determine whether a pod is ready. Pods that are ready will be part of the active service endpoints. A readiness probe determines whether a pod can service requests from clients. The way that the readiness check is performed on the pod is specific to what's running inside the pod. Kubernetes supports three kinds of readiness or liveness probes:

- *Exec probe*—Executes a command, and the exit code determines the result
- *HTTP Get probe*—Sends an HTTP GET request, and the HTTP status code determines the result
- *TCP Socket probe*—Attempts to establish a connection via TCP to determine the result

Using HTTP checks is a common technique for checking the health of applications. Spring Boot and WildFly Swarm support this out of the box from their actuator and monitor modules.

The source code contains this example in the chapter18/kubernetes/hello-swarm2 directory. In this example, we've added the org.wildfly.swarm:monitor dependency to the Maven pom.xml file. WildFly Swarm provides a health endpoint on /health that returns an HTTP 2xx status code if WildFly is healthy.

You can use this to know when WildFly Swarm is ready. To add the readiness probe in your microservice, you can add it to the Kubernetes manifest file. This can be done by creating a

deployment.yaml file in the src/main/fabric8 directory with the content from the following listing.

Listing 18.3 Kubernetes manifest with readiness and liveness probes

```
spec:  
  template:  
    spec:  
      containers:  
        - env:  
          livenessProbe: ①
```

①

HTTP liveness probe

```
    httpGet:  
      path: /health  
      port: 8080  
      scheme: HTTP  
      initialDelaySeconds: 30  
    readinessProbe: ②
```

②

HTTP readiness probe

```
    httpGet:  
      path: /health  
      port: 8080  
      scheme: HTTP  
      initialDelaySeconds: 10
```

As you can see, the Kubernetes manifest file can be nested deeply, such as five levels down, before you specify the liveness ① and readiness ② probes. Notice that you specify the HTTP URL to use, which is deferred as localhost (for example, <http://localhost:8080/health> is the URL that Kubernetes will periodically call).

Automatic health checks for Spring Boot and WildFly Swarm projects

The fabric8 Maven plugin can automatically add default readiness and liveness probes for specific Maven projects such as WildFly Swarm and Spring Boot. This happens automatically when a specific dependency is declared in the pom.xml file. For WildFly Swarm, you add the `org.wildfly.swarm:monitor` dependency, and with Spring Boot it's the `org.springframework.boot:spring-boot-actuator` dependency.

That's all that's needed. The example from hello-swarm2 with the health check configuration as shown in [listing 18.3](#) is unnecessary. You can find examples with the source code in the chapter18/hello-swarm3 and chapter18/client-spring-health directories. The fabric8-maven-plugin website provides additional documentation: <https://maven.fabric8.io>.

When the WildFly Swarm container is started, Kubernetes will wait 10 seconds before performing periodic readiness checks. If the container reports it's ready, it's added to the list of active endpoints on the service. But if anytime later the readiness check fails, the container is removed from the active endpoints and will no longer receive requests. If the container becomes active again, it will be re-added as an active service endpoint.

A liveness check is also periodically triggered by Kubernetes (both checks will keep running). If a liveness probe fails, Kubernetes will assume that the container is unhealthy and restart the container, or if deemed necessary re-create the pod.

This is an important difference between readiness and liveness probes. Readiness probes make sure only ready containers will receive traffic, whereas liveness probes keep the pods running by killing unhealthy containers and replacing them with healthy

ones.

You've now added both readiness and liveness probes to your WildFly Swarm microservice. You then deploy this improved microservice in the Kubernetes cluster:

```
cd chapter18/kubernetes/hello-swarm2  
mvn fabric8:deploy
```

After the pods have been redeployed, only one WildFly Swarm pod exists, because the Kubernetes manifest used by fabric8:deploy by default uses `replicas=1`.

SCALING UP THE POD WITH READINESS AND LIVENESS PROBES

When you scale up the WildFly Swarm deployment, you'd expect the readiness probe to be in use and the new pod to start to receive traffic only when it's ready. The Spring Boot client shouldn't log any errors while it continuously runs and calls the service. Let's see what happens:

```
$ kubectl scale deployment helloswarm-kubernetes --  
replicas=2
```

While the deployment is being scaled up, you can watch the pods using the `-w` flag:

```
$ kubectl get pods -w  
NAME                                     READY   STATUS  
RESTARTS   AGE  
helloswarm-kubernetes-2700449218-5fh1w   1/1    Running  
0          2h  
helloswarm-kubernetes-2700449218-wh2vk   0/1    Running  
0          7s  
spring-kubernetes-2151443245-27s8g       1/1    Running  
0          4h  
NAME                                     READY   STATUS  
RESTARTS   AGE  
helloswarm-kubernetes-2700449218-wh2vk   1/1    Running  
0          9s
```

While the pod is being started, the `READY` column is listed as `0/1`, which means the pod has one container and currently none is

ready. Then a little while later, the state changes and the column becomes 1/1, which means the container is ready.

When this happens, the Spring Boot client should start to load-balance its requests between the two pods and log responses from both pods:

```
Swarm says hello from helloswarm-kubernetes-2700449218-  
5fh1w  
Swarm says hello from helloswarm-kubernetes-2700449218-  
wh2vk
```

Are our microservices now resilient and fault-tolerant? Frankly, they're not. In the Spring Boot client, errors that you must deal with can still happen. You can see a problem in the client if you, for example, kill one of the running WildFly Swarm pods.

Using the Docker CLI tool, you can find the running Docker containers in the Kubernetes cluster and find the container ID for one of the running WildFly Swarm containers. You'll then kill this container and observe the Spring Boot client and see what happens:

```
docker kill 845ab817199f
```

You may have to attempt this multiple times to get to a situation where the client still fails when calling the service. We had to try three times before getting this error:

```
java.net.ConnectException: Cannot connect to  
10.0.0.199:8080
```

But the error should happen only one or two times, because the Kubernetes cluster is quick to detect that the WildFly Swarm pod has died and therefore isn't ready to service traffic from the client. The subsequent logs are all successful responses from the same pod, until later when both pods are up and running and the log shows responses from both of them:

```
Swarm says hello from helloswarm-kubernetes-2700449218-  
5fh1w  
Swarm says hello from helloswarm-kubernetes-2700449218-  
5fh1w
```

```
...
Swarm says hello from helloswarm-kubernetes-2700449218-
wh2vk
```

Your microservices are almost resilient and fault-tolerant. Building distributed systems such as a microservice architecture is a complex task, and you should always design your applications with failure in mind.

We previously covered this topic in chapter 7, section 7.4. In that chapter, you learned how to use Camel’s error handler or the Circuit Breaker EIP pattern to deal with failures when calling downstream services.

18.5.3 DEALING WITH FAILURES BY CALLING SERVICES IN KUBERNETES

In a distributed system, failures can happen anywhere, even when using Kubernetes. For example, the Spring Boot client can fail while calling the service from the WildFly Swarm application (which you’ve seen on several occasions so far in this chapter). You can reduce the chances for failure by replicating the downstream services to run on multiple pods and let Kubernetes load-balance among the active services. But even so, failures can still happen. For example, a container may just have died before Kubernetes has detected it’s no longer live or ready to handle requests.

You should design for failures that can and will happen. The good news is that Camel provides a rich set of error-handling functionality. For example, you can use Camel’s error handler to retry the failed service call:

```
errorHandler(defaultErrorHandler()
    .retryAttemptedLogLevel(LogLevel.INFO)
    .maximumRedeliveries(5)
    .redeliveryDelay(3000));

from("timer:foo?period=2000")
    .to("netty4-http://{{service:helloswarm-kubernetes}}")
    .log("${body}");
```

The error handler has been configured to retry up to 5 times and with a 3-second delay between attempts. To be able to see in the log when Camel performs a retry, you've raised the logging level to `INFO`.

As usual, we've provided an example with the source code. You can find this example in the `chapter18/kubernetes/client-spring2` directory.

The updated client can be deployed to the Kubernetes cluster as follows:

```
mvn fabric8:deploy
```

With our recent improvements in the WildFly Swarm microservice using the readiness and liveness probe, it's become much tougher to provoke failures when the client calls the service. Therefore, you'll roll back the WildFly Swarm deployment to use the older version that doesn't include a readiness probe. This can be done by deploying the older version:

```
cd chapter18/kubernetes/hello-swarm
mvn fabric8:deploy
```

You should now have only one pod running the WildFly Swarm microservice:

```
$ kubectl get pods
NAME                               READY   STATUS
RESTARTS   AGE
helloswarm-kubernetes-2700449218-tmw68   1/1    Running
1          2h
spring-kubernetes-1274767787-czg6s       1/1    Running
0          2h
```

You can provoke a failure by killing the WildFly Swarm pod, which causes Kubernetes to restart the container. Because it lacks a readiness probe, the container will receive traffic it can't process, and the client will either receive a connection exception or HTTP error codes.

The client logs the following:

```
Swarm says hello from helloswarm-kubernetes-2700449218-tmw68
On delivery attempt: 0 caught: java.net.ConnectException:
Cannot connect to 10.0.0.199:8080
On delivery attempt: 1 caught: java.net.ConnectException:
Cannot connect to 10.0.0.199:8080
Swarm says hello from helloswarm-kubernetes-2700449218-tmw68
Swarm says hello from helloswarm-kubernetes-2700449218-tmw68
```

As you can see, two unsuccessful attempts fail with a connection exception. The third time is the lucky charm, with successful responses coming back again.

In microservice architectures, the Circuit Breaker pattern is a popular choice for dealing with failures when calling services.

USING HYSTRIX AS CIRCUIT BREAKER

The Spring Boot client can easily be changed to use Hystrix as the circuit breaker (we covered Hystrix in chapter 7, section 7.4.4). All it takes is adding camel-hystrix-starter to the Maven pom.xml file, and changing the Camel route, as shown in the following listing.

Listing 18.4 Using Hystrix Circuit Breaker to deal with failures when calling a Kubernetes service

```
from("timer:foo?period=2000")
    .hystrix() ①
```

①

Hystrix EIP

```
    .to("netty4-http:http://helloswarm-
kubernetes:8080") ②
```

②

Calling Kubernetes service using its DNS name

```
.onFallback()
```

③

③

Fallback if the service call failed

```
.transform().constant("Cannot call downstream  
service")  
.end()  
.log("${body}");
```

The Camel route is easy to understand. You've protected calling the downstream service in Kubernetes using Hystrix ①. We've taken the opportunity to showcase another great feature in Kubernetes: you can call Kubernetes services using DNS. The hostname is simply the service name, so the WildFly Swarm service can be called using <http://helloswarm-kubernetes:8080> ②. In case calling the downstream service fails, the Hystrix fallback ③ is executed, which sets a constant message body that will be logged.

Hystrix will by default use a 1-second time-out when calling a downstream service. This threshold may often be too low, and you can configure a higher value. When using Spring Boot, you can also configure Camel Hystrix from the application.properties file, which is easy. To set a time out of 2 seconds, all you have to do is add the following line:

```
camel.hystrix.execution-timeout-in-milliseconds=2000
```

You can find this example in the chapter18/kubernetes/client-spring3 directory. To try this, you need to deploy the updated client:

```
mvn fabric8:deploy
```

At this point, you still have only one pod of the WildFly Swarm microservice running, so it's easy to provoke a failure by deleting the pod:

```
kubectl delete pod helloswarm-kubernetes-2700449218-tmw68
```

And you can see that the Spring Boot client should start to fail:

```
Swarm says hello from helloswarm-kubernetes-2700449218-tmw68
Cannot call downstream service
Cannot call downstream service
...
```

After a little while, the WildFly Swarm pod has been created again, and the client starts logging successful responses:

```
Swarm says hello from helloswarm-kubernetes-2700449218-1lv13
```

Phew! That was a lot to take in. Let's recap what you learned.

WHAT DID YOU LEARN?

Building resilient and fault-tolerant microservices is a key requirement in distributed systems. You have to face the music and figure out how to design applications that can survive in an environment where failures will happen.

You learned that Kubernetes provides great support for this with services, replication controllers, and deployments that can help ensure that the cluster can self-heal. But developers still have to design their microservices with failures in mind. For example, microservices that provide services must implement some kind of readiness check allowing Kubernetes to route traffic only to containers that are ready to handle requests. Likewise, you learned that liveness checks must be implemented in your microservices so Kubernetes can constantly observe your applications and automatically restart or kill containers that are no longer alive.

Even with all this awesomeness from Kubernetes, the clients that are calling services must accept that failures can happen. When using Camel, you can use Camel's error handler or integration with Hystrix to elegantly handle those failures.

Unit and integration tests are a vital part of the lives of developers. You may wonder how you perform testing with

Kubernetes. That's a good question, so let's spend a moment to cover the basics of testing Kubernetes.

18.6 Testing Camel microservices on Kubernetes

This section shows you how to unit-test Camel on Kubernetes. You're going to use the Arquillian testing framework. (Arquillian was also used in chapter 9.) More precisely, you'll use the Arquillian Cube (<http://arquillian.org/arquillian-cube>) extension, which provides capabilities for managing and running tests on Docker and Kubernetes.

In this section, you'll first add Arquillian Cube to your project. Then you'll write a basic unit test and run the test in Kubernetes.

18.6.1 SETTING UP ARQUILLIAN CUBE

You'll use the Camel microservice that runs on Spring Boot, which is the simplest project. This example is located in the chapter18/kubernetes/client-spring-test directory.

You add Arquillian Cube to any Java project via Maven by importing the arquillian-cube-bom BOM to the dependency management of your Maven pom.xml file. Then add the following four Maven dependencies: arquillian-junit-standalone, arquillian-cube-kubernetes, assertj-core, and kubernetes-assertions. The last two dependencies aren't from Arquillian but make writing assertions using Kubernetes concepts much easier (you'll see this in a moment). The last thing to do is to create an arquillian.xml file in the src/main/resources directory with the following content:

```
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="http://jboss.org/schema/arquillian"
            xsi:schemaLocation="http://jboss.org/schema/arquillian
                http://jboss.org/schema/arquillian/arquillian_1_0.xsd">
    <extension qualifier="kubernetes">
```

```
</extension>  
</arquillian>
```

You do this to turn on the Kubernetes extension.

18.6.2 WRITING A BASIC UNIT TEST USING ARQUILLIAN CUBE

The following listing shows a unit test that we highly recommend you run. The test checks that the application (pod) can start and be ready for a stable period of time (10 seconds).

Listing 18.5 Testing that the pod can start and run for a stable period of time

```
@RunWith(Arquillian.class)  
@RequiresKubernetes ①
```

①

Test requires Kubernetes

```
public class PodStartedKT {
```

```
    @ArquillianResource  
    KubernetesClient client; ②
```

②

Injects Kubernetes client

```
    @Test  
    public void testPodStarted() throws Exception {  
  
        assertThat(client).deployments().pods().isPodReadyForPeriod()  
(); ③
```

③

Asserts pod can deploy and be ready

```
} }
```

As you can see, the test has only a few lines of source code. The unit test uses Arquillian with Kubernetes ❶. Notice that you name the class `PodStarterKT` with the suffix `KT` to denote that this is a Kubernetes Test. `KubernetesClient` is injected as a resource ❷, which is being used in the test method that's only one line of code. That single line of code ❸ asserts that the deployment of the pod is ready for a stable period (10 seconds).

This is a powerful test because it tests that your application can be deployed, can start, and can run on Kubernetes.

18.6.3 RUNNING ARQUILLIAN CUBE TESTS ON KUBERNETES

You can run this test from a command line:

```
eval $(minikube docker-env)
cd chapter18/kubernetes/client-spring-test
mvn install -P kubernetes
```

Notice that you must specify `-P kubernetes` as an argument. This allows you to build and run regular unit tests locally without Kubernetes separated from running the Kubernetes test. You can find more details about how this is done in the `pom.xml` file of the example.

When the test runs, it logs to the console a bit of activity that happens and then completes with the following:

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 34.841 sec
- in com.camelaction.PodStartedKT
Destroying Session:a83be678-55d1-4888-b3a1-845bc0aef455
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Here you can see that one test ran and completed with success in about 34 seconds. In that time, Arquillian Cube first uses the `KubernetesClient` to create a new temporary namespace in Kubernetes. Using a temporary namespace, you ensure that the unit test runs in a clean environment and doesn't cause side effects to any other pods and services running in the Kubernetes

cluster. Then the application is deployed, and the test waits for the pod to become ready and stay ready for at least 10 seconds to be considered stable. When this happens, the test is complete and the pod is undeployed, and the temporary namespace is deleted.

Okay, that's impressive, but maybe you don't believe us. Let's, as an example, make a mistake in our Camel application on purpose and rerun the tests. In `HelloRoute`, change the route to use a nonexistent Camel component, which should cause Camel to fail on startup; therefore, we'd expect the pod to not become ready. We change the route from using `timer` to `timer2` as the component name:

```
from("timer2:foo?period=2000")
```

And then rerun the tests:

```
mvn install -P kubernetes
```

This time, the test takes longer to complete and should fail:

```
Tests in error:  
  PodStartedKT.com.camelinaction.PodStartedKT  
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0
```

When a test fails, Arquillian Cube will grab the log from the failed pod and output it in the command shell. You can scroll up and find the error reported from Camel about the unknown component:

```
Failed to resolve endpoint: timer2://foo?period=2000 due  
to: No component  
found with scheme: timer2
```

This was a quick introduction to writing a basic unit test for Kubernetes. There's one more test we'd like to show you: a test that calls a Kubernetes service.

18.6.4 WRITING A UNIT TEST THAT CALLS A KUBERNETES SERVICE

In this section, you're going to write a unit test that calls the Hello service from the WildFly Swarm Camel example. You're going to use Arquillian Cube again, which you'll set up the same way as previously covered. The source code for the unit test is provided in the following listing.

[Listing 18.6](#) Unit test that calls service running inside Kubernetes

```
@RunWith(Arquillian.class)  
@RequiresKubernetes 1
```

1

Test requires Kubernetes

```
public class HelloSwarmKT {
```

```
    @ArquillianResource 2
```

2

Injects Kubernetes client

```
        KubernetesClient client; 2
```

```
        @Named("helloswarm-kubernetes") 3
```

3

Injects URL to Kubernetes service to be called

```
        @PortForward 3  
        @ArquillianResource 3  
        URL url; 3
```

```
    @Test
```

```
    public void testCallService() throws Exception {  
        OkHttpClient okHttpClient = new OkHttpClient();  
        Request request = new  
        Request.Builder().get().url(url).build();
```

```
        Response response =
```

```
okHttpClient.newCall(request).execute();
```

4

Uses OkHttpClient to call service

```
assertEquals(200, response.code());  
  
String body = response.body().string();  
assertTrue(body.contains("Swarm says hello  
from")); 5
```

5

Asserts response is as expected

```
}
```

The test requires running on Kubernetes, which is declared using the `@RequiresKubernetes` annotation on the class ①. The test will use the `KubernetesClient` that's being injected ②. The service you want to call in Kubernetes is then declared with a couple of annotations ③. The `@Named` annotation specifies the name of the service, and `@PortForward` ensures that the service is accessible from the JVM running the unit tests into the running Kubernetes cluster via Kubernetes port forwarding.

The service is then called using plain HTTP and the `okHttpClient` library. Notice that this library has no knowledge of Kubernetes but uses the injected URL as is. Because of the port forwarding, the `okHttpClient` is able to call ④ into the running Kubernetes cluster and receive the response, which is then asserted as expected ⑤.

You can try this example from the source code as follows:

```
eval $(minikube docker-env)  
cd chapter18/kubernetes/hello-swarm-test  
mvn install -P kubernetes
```

The test should complete successfully. As before, you can make a change to the source code to provoke an error and see what

happens. For example, you can try to change the response generated in `HelloBean` to say goodbye instead of hello:

```
public String sayHello() throws Exception {  
    return "Swarm says goodbye from " +  
InetAddressUtil.getLocalHostName();  
}
```

And then rerun the tests again:

```
mvn install -P kubernetes
```

This should fail with an assertion error because the test expects the response to say hello and not goodbye.

Speaking of goodbye, that's all we have to say about testing Kubernetes. The last section is a mixed bag of some great Kubernetes-related projects.

18.7 Introducing fabric8, Helm, and OpenShift

This section describes projects related to Kubernetes that are worth keeping an eye on. The first project is fabric8, which you've already used through its Maven tooling, such as the Docker and fabric8 Maven plugins. We will give you a brief overview and history of the fabric8 project and encourage you to take a closer look once in a while to see what the fabric8 team has done of late.

Linux users who are accustomed to installing software from package managers such as yum or apt will be pleased to know that Helm is their answer in the realm of Kubernetes. Before ending this chapter, we'll look at OpenShift, a platform built on top of Kubernetes, and see the additional capabilities that it brings to the table.

18.7.1 FABRIC8

The fabric8 project (<https://fabric8.io>) has evolved over the years. The project started as a platform to manage a cluster of

Apache Karaf or JBoss Fuse application servers. This was before Docker or Kubernetes existed, when we had to build our own solutions with pieces from Java such as Apache ZooKeeper, Maven, and OSGi. Docker and Kubernetes changed all that, and fabric8 v2 was completely redesigned to be a platform on top of Kubernetes.

Today, fabric8 is a pure upstream project with a high degree of creativity that enables new ideas to be born and attempted. The good parts of fabric8 are then harvested and bought into other projects and communities such as Kubernetes, OpenShift, OpenShift.io, Apache Camel, and Spring Cloud. For example, fabric8 was a first mover on a better Java developer story on top of Kubernetes. Other successes are the fabric8-maven-plugin and the Kubernetes Java client used by various tooling and projects to make it easier for Java developers to work with Kubernetes. The continuous deployment effort from fabric8 found its way into the OpenShift project.

Moving on to the next project. The purpose of Helm is to make installing applications on Kubernetes easy.

18.7.2 KUBERNETES HELM

Kubernetes Helm is like an OS package manager (yum or apt in Linux), but for installing and managing applications in Kubernetes. Helm provides a command-line tool with the surprising name of helm, and a server component named Tiller that runs as a pod inside Kubernetes. Helm application packages are called charts.

A *chart* is just a YAML file with overall details about the application. Kubernetes resources are embedded in the chart, and these resources can be templated. In other words, they can be parameterized, so you can install an application and take in parameters from the command line such as usernames, passwords, and database names.

This makes it easier to install well-known applications such as MySQL, PostgreSQL, WordPress, and others. The Helm

community maintains a growing list of charts at <https://github.com/kubernetes/charts>.

Let's try this.

INSTALLING HELM

To install Helm, you can follow the instructions at the Helm website: <https://github.com/kubernetes/helm>. There are downloads for macOS, Linux, and Windows.

After installing Helm, you need to install Tiller in the Kubernetes cluster, and you can easily do this with the helm CLI tool:

```
helm init
```

Then wait a little while for the Tiller pod to be ready. If you can't find the Tiller pods, it's because they're not installed in the default namespace but in the Kubernetes system:

```
$ kubectl get pod -n kube-system
NAME                               READY   STATUS    RESTARTS
AGE
kube-addon-manager-minikube      1/1     Running   2
1d
kube-dns-v20-x1hf8              3/3     Running   6
1d
kubernetes-dashboard-tx751      1/1     Running   2
1d
tiller-deploy-3210613906-f936w  1/1     Running   0
1h
```

After Helm is installed, you can use the CLI to search for applications to install. For example, type this to find any MySQL applications:

```
helm search mysql
```

To install MySQL, type the following:

```
helm install --name my-great-database stable/mysql
```

During the installation, Helm will output further instructions,

such as how to get generated passwords and URLs to the installed application. You can delete what you installed at any time by deleting the chosen name:

```
helm delete my-great-database
```

INSTALLING YOUR OWN APPLICATION USING HELM

You can try to install the two Camel microservices using Helm. Because fabric8-maven-plugin generates Helm charts, you can easily install the example. A prerequisite is that the Docker image must have been built and published to the Kubernetes Docker repository. This can be done as follows:

```
cd chapter18/kubernetes/hello-swarm2  
mvn install
```

You can now install the application using Helm:

```
helm install target/helloswarm-kubernetes-2.0-SNAPSHOT-  
helm.tar.gz
```

And likewise, you can install the Spring-Boot client:

```
cd chapter18/kubernetes/client-spring3  
mvn install  
helm install target/spring-kubernetes-2.0-SNAPSHOT-  
helm.tar.gz
```

Notice that the Helm charts a tar.gz file, which you can publish to a Maven repository. Helm also makes it possible to build your own Helm chart repositories—for example, all the applications you use in your company.

The fabric8 project provides a chart repository that you can add using the following:

```
helm repo add fabric8 https://fabric8.io/helm
```

Helm is a promising package manager for Kubernetes, and we recommend taking a look at it.

PLATFORMS BUILT ON TOP OF KUBERNETES

Kubernetes is a great container platform that's good at orchestrating and running containers at scale. As mentioned, Kubernetes was created by Google, based on its experience of running containers for over 10 years. And as such, Kubernetes is operation focused, and not as accessible for developers and system administrators from traditional enterprises. Something more is needed. That's why several companies are building platforms on top of Kubernetes that provide a better experience for developers and system administrators. Those platforms are known as *platforms as a service* (PaaS). One of these platforms is OpenShift by Red Hat.

18.7.3 OPENSHIFT

Red Hat OpenShift is a platform as a service built on top of Kubernetes. OpenShift comes in three offerings:

- *OpenShift Origin*—Open source upstream project that's free to use
- *OpenShift Enterprise*—Commercial enterprise-grade product by Red Hat that runs on premises or in any of the big cloud providers such as Amazon, Azure, or Google
- *OpenShift Online*—Online multitenant cloud hosted by Red Hat

OpenShift comes with a rich set of features and functionality out of the box. For example, a feature-rich web console makes using the platform better for both developers and system administrators.

OpenShift provides powerful user-management capabilities, making it possible to specify what each user can and can't do. Each user is given access to the projects they work on (a *project* is a Kubernetes namespace). A user can act on only those resources that reside in the projects the user has access to. Access to projects is granted by cluster administrators. OpenShift also allows you to integrate to existing user-management systems that organizations have, such as Microsoft Active Directory or any other LDAP servers.

BUILDING IMAGES FROM SOURCE

One of the most appreciated features of OpenShift is its ability to build, deploy, and run applications straight from source code. With Kubernetes, users have to build their Docker images themselves (as you did in this chapter with fabric8-maven-plugin and with Docker installed locally on your computer). OpenShift can pull the source code from a code repository such as GitHub and build it as a Docker image within the cluster. This can also be seen as a more secure option, because you'd be in full control of how your images are built, as it's all done within the platform. It can be a security concern to let developers run arbitrary Docker containers they've found on the internet.

OpenShift can integrate with web hooks. When new code is pushed in the code repository, a new build is triggered that, when the Docker image has been successfully built, can trigger a rolling upgrade of pods running with same image tag.

Figure 18.9 is a screenshot of the OpenShift web console with a rolling upgrade in progress.

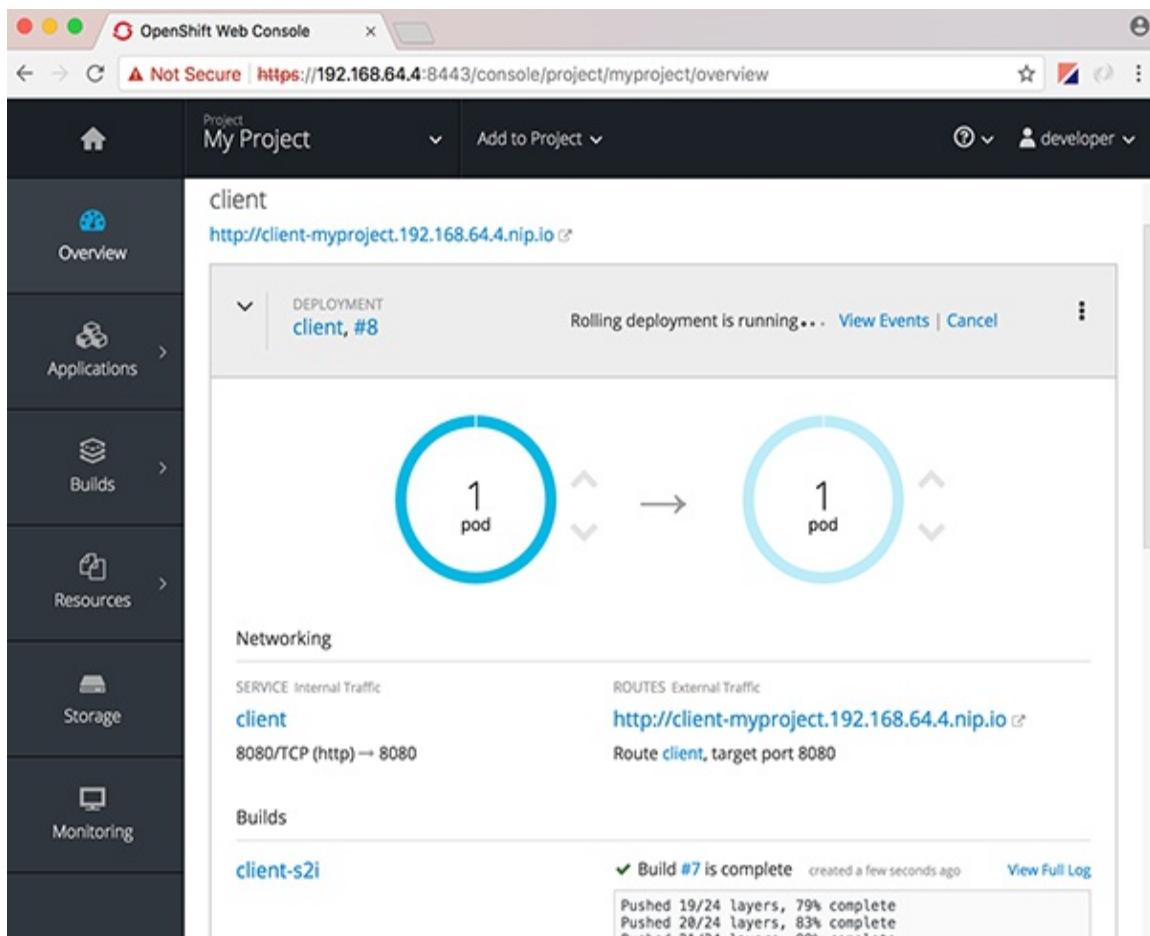


Figure 18.9 OpenShift web console. A source code change is pushed to GitHub, which triggers a build running in the cluster, which on success pushes an updated Docker image to the OpenShift Image Repository. The updated image then triggers a rolling upgrade of all pods that run this image, which is what is in progress on the screen. The old image is currently running and waiting for a new pod to spin up with the updated image.

Continuous delivery is also provided out of the box; you can customize your pipelines to your business processes. For example, before changes can go into production, an approval process is triggered, and a manager must accept the change before the pipeline continues.

TRYING OPENSHIFT

If you want to try OpenShift, you can install it locally using Minishift, which is equivalent to Minikube. Another way of trying OpenShift is by accessing OpenShift Online, which is a hosted platform that offers a free plan with numerous pods and

resources. You can purchase additional resources if needed.

If you're going down the path of OpenShift, we highly recommend that you download the free ebook *OpenShift for Developers* (the link is provided in this chapter's summary), which has more details about the differences between Kubernetes and OpenShift and ways to get started developing on this platform.

18.8 Summary and best practices

This chapter is an introduction and hands-on guide to getting started with Camel in the world of Docker and Kubernetes. There's so much more to this new world of containers that warrants several books on its own. This chapter taught you how to get your Camel and Java applications running on Kubernetes. Although Docker is only four years old (founded in 2013), the concept of Linux containers dates back decades. Still, using containers has only recently started to become more mainstream. As developers, we have to adapt and learn the skills required to stay relevant in our industry. We encourage you to start playing with containers when you get a chance.

The takeaways from this chapter are as follows:

- *Learn about containers*—The concept of containers and their impacts on the way we package, deploy, and run our applications at scale are here to stay. Climb onboard and learn the skill set for this new world.
- *Camel, containers, and microservices for the win*—Camel runs great on containers and as a microservice library. You can use Camel in popular microservice frameworks such as Spring Boot or WildFly Swarm.
- *Learn the Kubernetes concepts*—Kubernetes is the third-generation cluster platform from Google, which has more than a decade of experience running everything in containers. Kubernetes is a goldmine of great concepts for a distributed

system.

- *Learn using Minikube*—Get started with Kubernetes on your own computer using Minikube. Learn how to build and run your Java applications in this local cluster. Don’t be afraid, because if something goes wrong, you can always stop and delete Minikube and start from scratch.
- *Use fabric8-maven-plugin*—Developers using Maven should use this plugin to help them build, run, and debug their Java applications running in Kubernetes.
- *Design for failure*—Failures will happen in a distributed system. Build your Java applications with this in mind. Use the readiness and liveness probes from Kubernetes to make your applications more fault-tolerant, and use Kubernetes services for network communications.

We’ve covered a lot in this chapter but certainly didn’t cover everything. When using Kubernetes (or any other container-based platform), there are many more things to consider in a microservices environment that we can’t cover in this book. We don’t want to leave you in despair, so we’ve listed some things you should consider here:

- *Logging and metrics*—Make your applications observable so a centralized platform-monitoring solution can capture logs and metrics. We covered monitoring and management in chapter 16. PaaS platforms such as OpenShift come with such functionality out of the box.
- *Tracing*—The more you break your applications into individual microservices, the more moving parts you have. You need tooling to help you see the big picture. For example, use message tracing such as Zipkin or OpenTracing so you can correlate a business transaction with other data from logging, metrics, and response times.
- *Continuous delivery*—When you have many more microservices than before with traditional architecture, your existing manual

deployment process won't scale. Teams that own and operate their own microservices also need a way of deploying without intervention from a centralized operations team. For this, you need to use continuous deployment, which can automate deployment of applications into every environment in the cluster.

- *Configuration management*—Your microservices should be configurable—for example, to configure sensitive login credentials, or to turn on features for A/B testing. In Kubernetes, you can use *secrets* and *configmaps* to configure your running containers.
- *Enterprises should consider using a PaaS*—Companies looking to adopt Kubernetes may benefit from using a full-fledged PaaS platform to provide functionality that enterprises need and expect. Using vanilla Kubernetes often requires you to find and integrate needed third-party add-ons that you can otherwise spend on delivering value to your business.

Here are a few books and other resources that we found valuable when we were learning about Docker and Kubernetes:

- “The Decline of Java Application Servers when Using Docker Containers” by James Strachan (<https://blog.fabric8.io/the-decline-of-java-application-servers-when-using-docker-containers-edbe032e1f30#.1n6jphq5y>)
- *Kubernetes in Action* by Marko Luksa (Manning, 2017)
- *Kubernetes Patterns* (<https://leanpub.com/k8spatterns/>) by Bilgin Ibryam and Roland Huss (Leanpub, 2018)
- *Microservices for Java Developers* by Christian Posta (free book can be downloaded from <https://developers.redhat.com/promotions/microservices-for-javadevelopers/>)
- *OpenShift for Developers* by Grant Shipley and Graham Dumpleton (free book can be downloaded from www.openshift.com/promotions/for-developers.html)

We'll now leave the world of containers and talk about something completely different. Chapter 19 is dedicated to Camel tooling. We'll cover what we believe are the greatest Camel tools you can find; these come either out of the box from Apache Camel or from third-party projects and vendors.

19

Camel tooling

This chapters covers

- Eclipse-based graphical Camel editor from JBoss
- IDEA plugin for Camel code editor
- Maven plugin for source code validation
- Debugging Camel routes from Eclipse or a web browser
- hawtio web console with Camel plugin
- Camel Catalog—information archive for Camel tools

This chapter walks you through some of the most powerful Camel tools that you can find on the internet. The first tool covered is an Eclipse plugin that provides a graphical editor that enables users to design Camel routes using a drag-and-drop style of editing. For IDEA users, we have an alternative tool that allows in-place code assistance for editing Camel endpoints in a *type-safe* manner. We'll show you a Maven tool that's capable of scanning your source code and reporting any Camel configuration mistakes. You can also find tools to debug Camel applications by stepping through the routes and inspecting the Camel messages.

The last tool we demonstrate can inspect running Camel applications and help you visualize the Camel routes with real-time metrics. It also provides management functionality that

allows you to control your Camel applications—for example, starting and stopping routes. You’ll learn the secret of the Camel Catalog, which exposes a wealth of information about a Camel release that allows tooling to know everything there is to know about all the components, data formats, EIPs, and so on.

Let’s get started with the traditional developer tools that build Camel routes using a graphical editor.

19.1 Camel editors

Building Camel tools for graphical Camel development is a large task and could explain why we see this kind of tool only from commercial vendors such as Red Hat and Talend. This chapter covers the tools we know best: those from Red Hat JBoss, Apache Camel, and hawtio projects.

19.1.1 JBOSS FUSE TOOLING

The JBoss Fuse Tooling (formerly known as Fuse IDE—see <http://tools.jboss.org/features/fusetools.html>) is a plugin that can be installed in Eclipse and that provides the following Camel development capabilities:

- Graphical route editor
- Graphical data mapper
- Type-safe endpoint editor
- Integrated Camel tracer and debugger
- Easy deployment to application servers

No vendor lock-in

The tools covered in this chapter are all open source and have no vendor lock-in. You can start to use a tool and can stop at any point. These tools all operate on your project’s

source code as is and don't use tooling-specific metadata to be saved with your source code.

We'll now look at some of the features this tool provides. You can find the sample project in the chapter19/eclipse-camel-editor directory.

GRAPHICAL CAMEL EDITOR

The Camel project that's loaded into Eclipse in [figure 19.1](#) is a simple Camel route from a Spring XML file. You can click each EIP in the route and edit its properties in a type-safe manner, as shown in [figure 19.2](#).

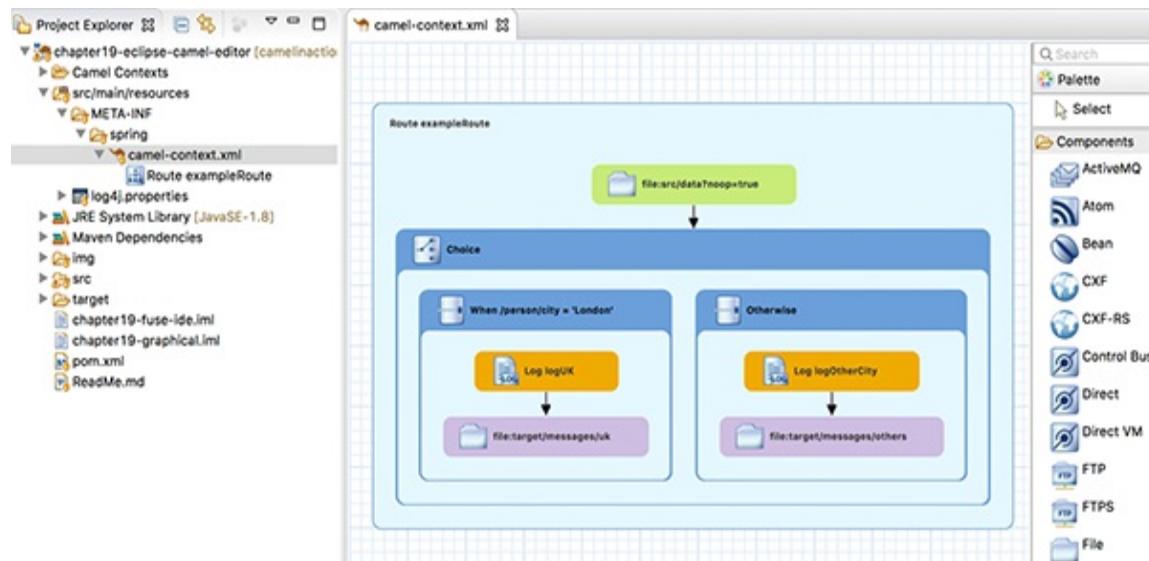


Figure 19.1 Graphical Camel editor showing the Camel route using EIP icons and depicting their connections. Clicking an EIP allows you to configure the EIP in the properties panel in a type-safe way.

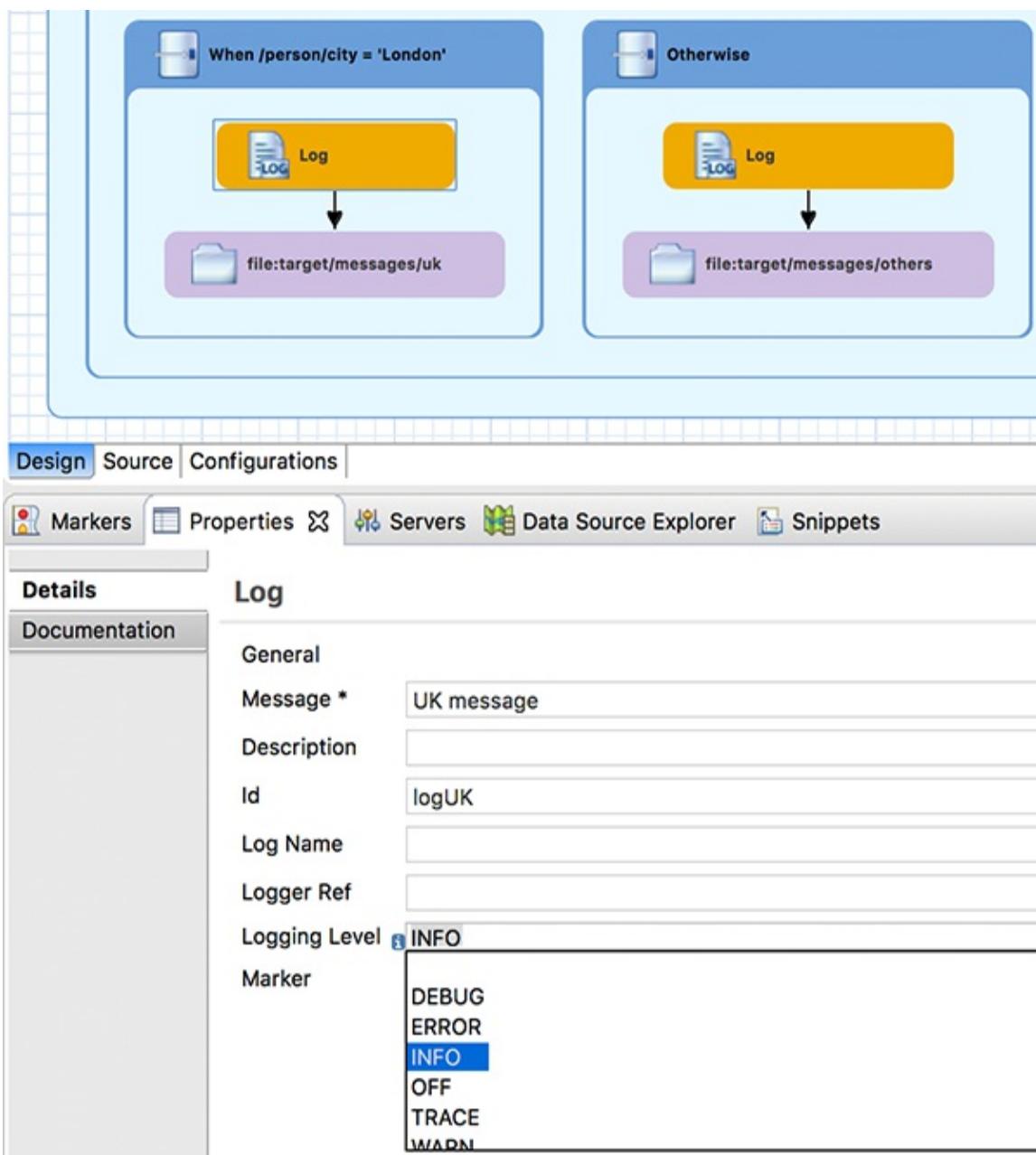


Figure 19.2 Type-safe editing the selected Log EIP from the route in the panel underneath. The drop-down panel for the logging level shows the possible values accepted.

In figure 19.2, we've selected the Log EIP; the tiny blue box around its borders indicates the selection. In the properties panel underneath, the possible options from the Log EIP are shown with its current values. If you want to change the logging level, you can click the drop-down, which lists the possible values as shown in the screenshot.

TYPE-SAFE ENDPOINT EDITOR

The tooling also offers type-safe editing of all the Camel endpoints. The example from chapter19/eclipse-camel-editor is a Camel route that consumes files from the following endpoint:

```
<from uri="file:src/data?noop=true"/>
```

Now, suppose you want to configure a read lock on this endpoint. Using the tool, you have all the options available presented in the editor. By hovering the mouse pointer over an option's label, you can access the documentation as a tooltip. Figure 19.3 shows how you're changing the endpoint to use the changed read lock with a 5-second interval.

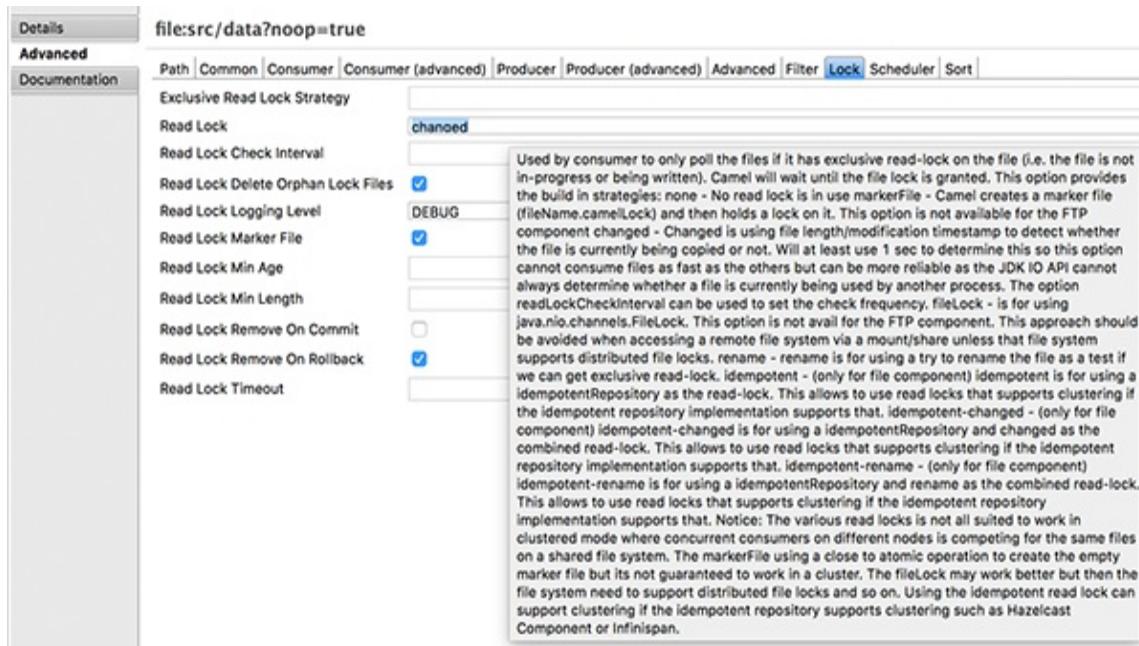


Figure 19.3 Editing the Camel endpoint in a type-safe manner using the tooling. Documentation is available for every option as tooltips. The screenshot shows the extensive documentation for the read lock option.

After editing, the endpoint is updated accordingly:

```
<from uri="file:src/data?  
noop=true&readLock=changed&readLockCheckInterval=50  
00"/>
```

INTEGRATED CAMEL TRACER AND DEBUGGER

The tool provides an Eclipse-based Camel debugger. To try this in action, right-click one or more EIPs in the graphical editor and click the Set Breakpoint menu item. The EIP icon should show a red dot indicating that the breakpoint is set. Then right-click the camel-context.xml file, as shown in [figure 19.4](#), and choose Debug As > Local Camel Context (Without Tests).

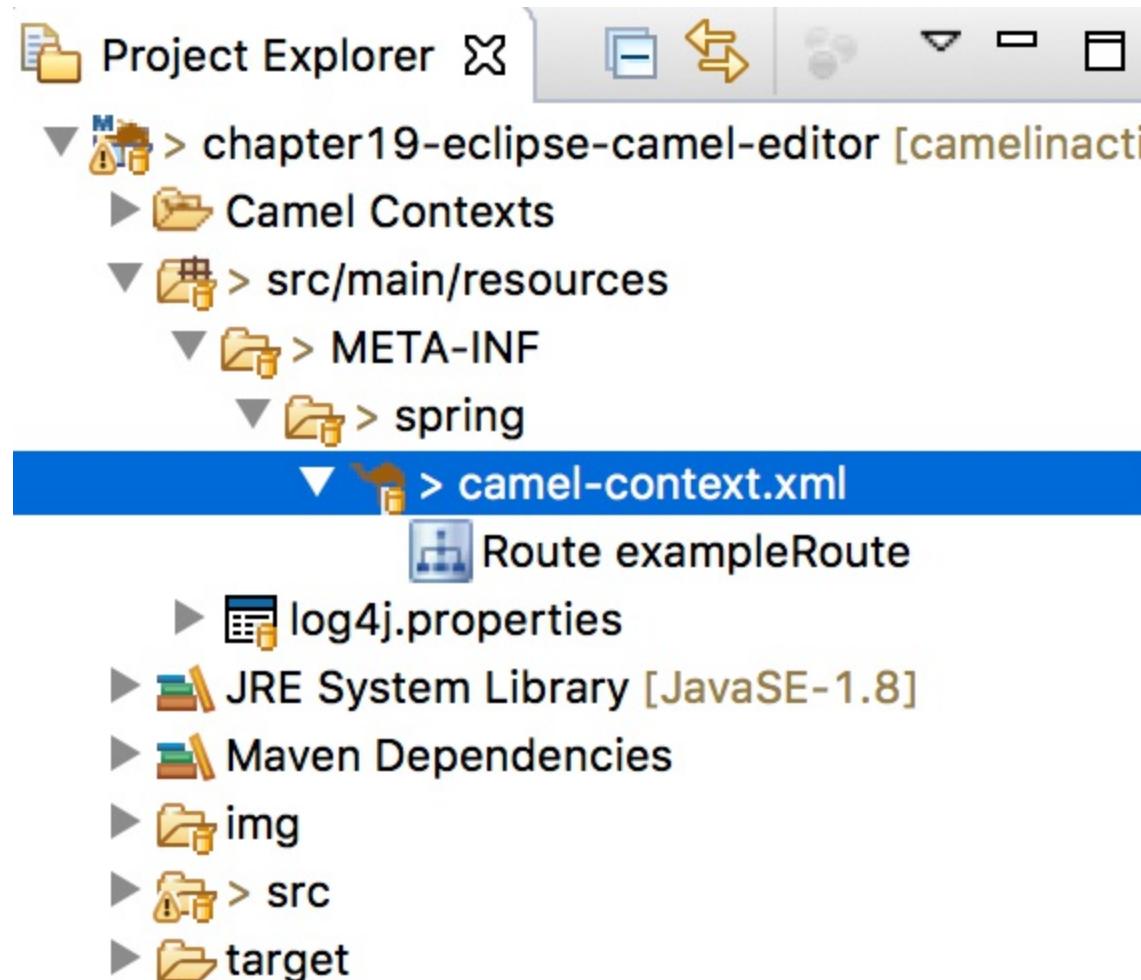


Figure 19.4 To debug a Camel application, right-click the XML file containing `<camelContext>` (outlined here) and choose Debug As > Local Camel Context (Without Tests) to start the application in debug mode.

Eclipse then switches to debug perspective, and when a message hits the breakpoint, the EIP icon will be highlighted with a solid red border, as shown in [figure 19.5](#).

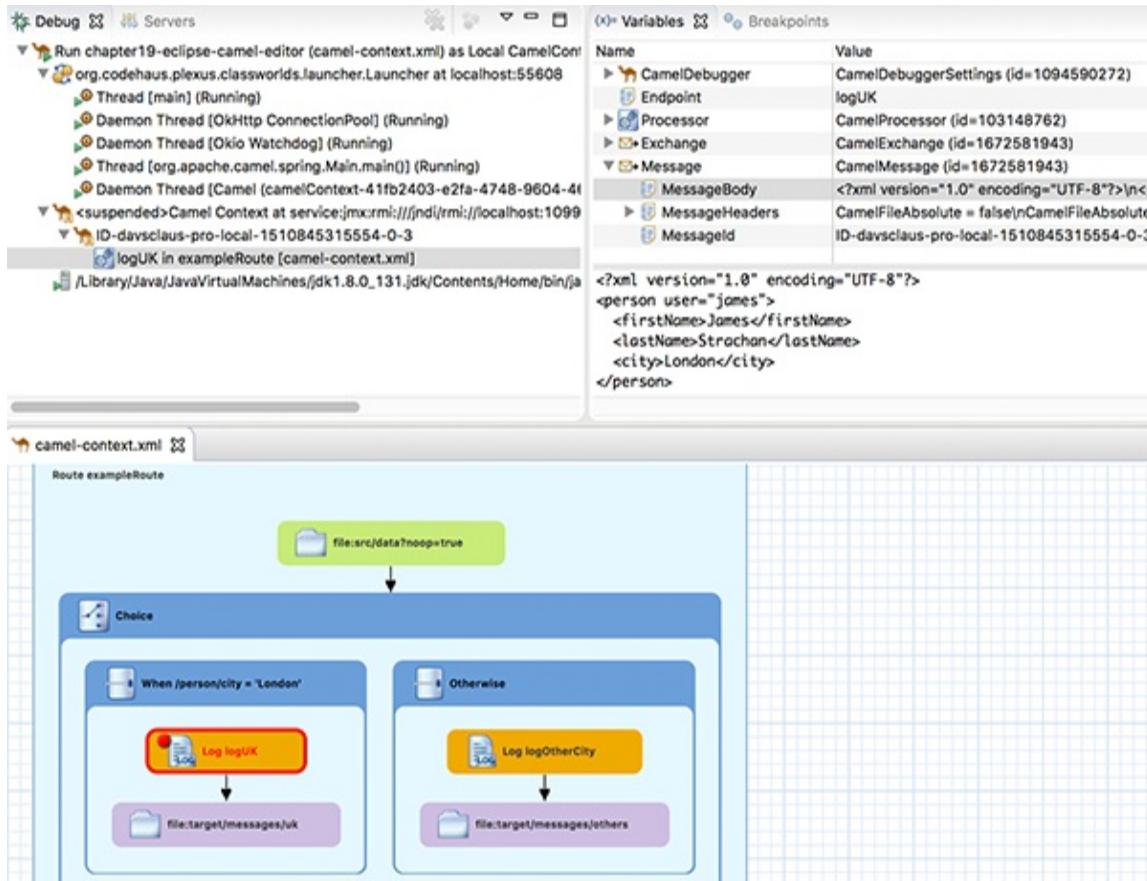


Figure 19.5 Debugging a Camel application in Eclipse: a Camel message hits the breakpoint highlighted by an outline. In the top-right corner, the Variables tab shows the content of the message. We've selected the message body, shown underneath, and in this example is an XML message showing that James Strachan lives in London (he lived there while creating Apache Camel).

You can use the tooling to inspect the message content, such as the message body and its headers. You can even update the message body from the debugger, such as changing the content of the message body or headers. When you're ready to continue, click either the Resume or Step Over button. The latter allows you to single-step through the route, stopping at the next EIP, as shown in figure 19.6.

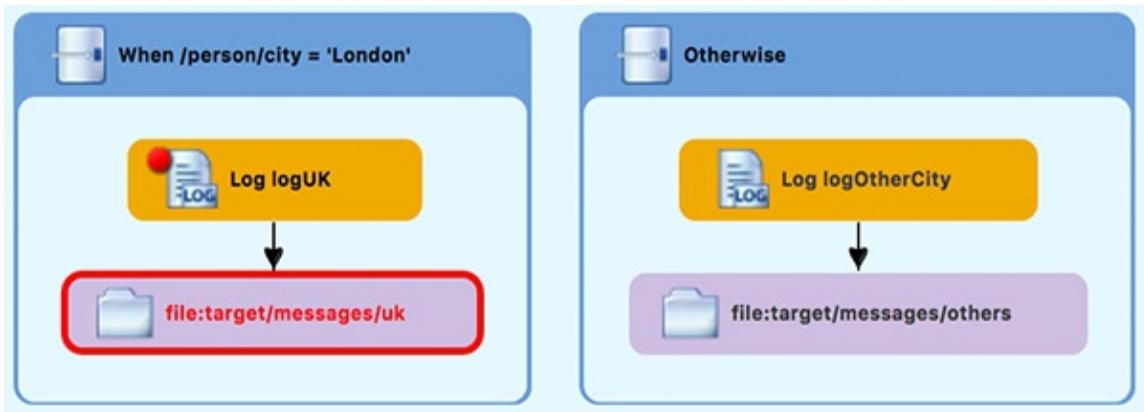


Figure 19.6 The breakpoint was set on the Log EIP indicated by the dot in the upper left corner. When the breakpoint was hit, we clicked the step over button (F6), and the debugger ran to the next EIP, which is the file endpoint indicated by the square.

Table 19.1 provides a summary of JBoss Fuse Tooling.

Table 19.1 The good and not so good about JBoss Fuse Tooling

Pros	Cons
Graphical route editor (only works with XML DSL)	Works only with Eclipse.
Powerful Camel debugger natively integrated in Eclipse	The graphical Camel debugger only works with XML DSL.
Type-safe Camel endpoint editor (currently only works with XML DSL)	
Documentation for all components and EIPs included	
Full-time engineers working on the tool	
100% open source and Eclipse licensed	

Let's look at another Camel editor that works in a different way.

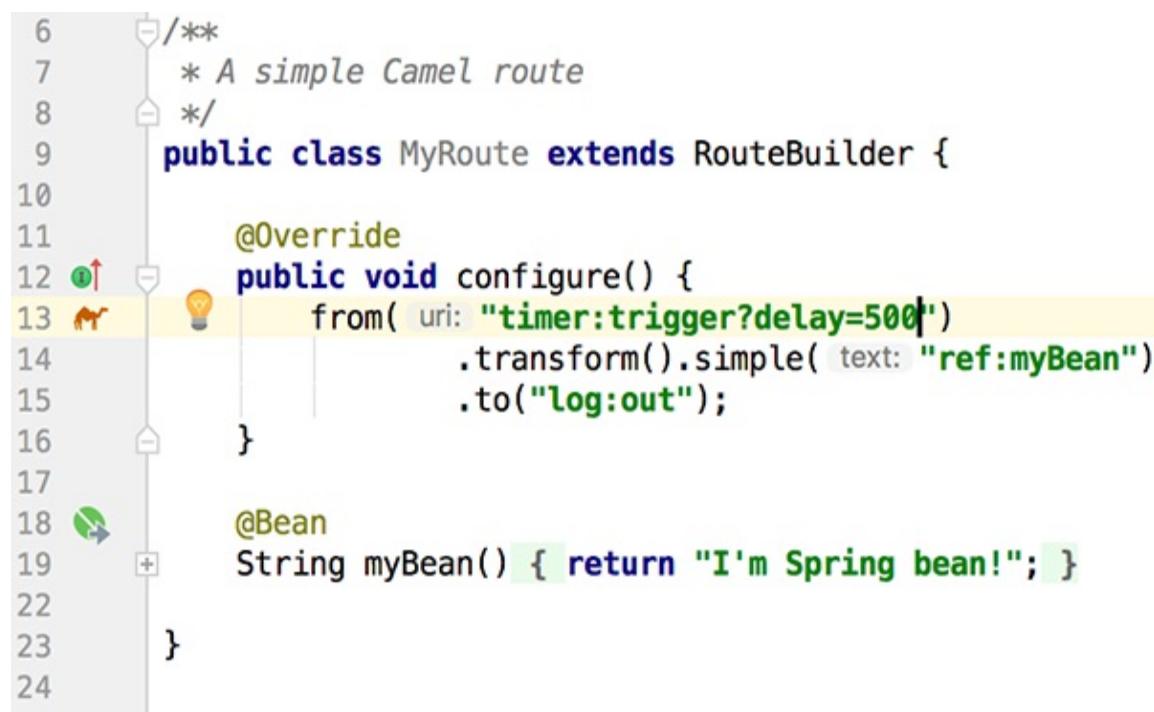
19.1.2 APACHE CAMEL IDEA PLUGIN

IntelliJ IDEA users shouldn't miss out on the excellent Apache Camel IDEA plugin. This plugin adds Camel awareness to the IDEA editor to bring great power and enjoyment for developers

to work on Camel code. The plugin is best explained by just trying it out in action, so let's do that.

To install the plugin, start IDEA and open its preferences. From within Preferences, select Plugins and click Browse Repositories. In the search field, type Apache Camel to find the Apache Camel IDEA plugin, which you then select. Click Install. After installation, you may need to restart IDEA.

From the book's source code, you can open the chapter19/camel-idea-editor example in IDEA. Then open the Camel route file, MyRoute.java, and you should notice the subtle Camel plugin appearance in the code editor by small Camel icons in the gutter, as illustrated in [figure 19.7](#).



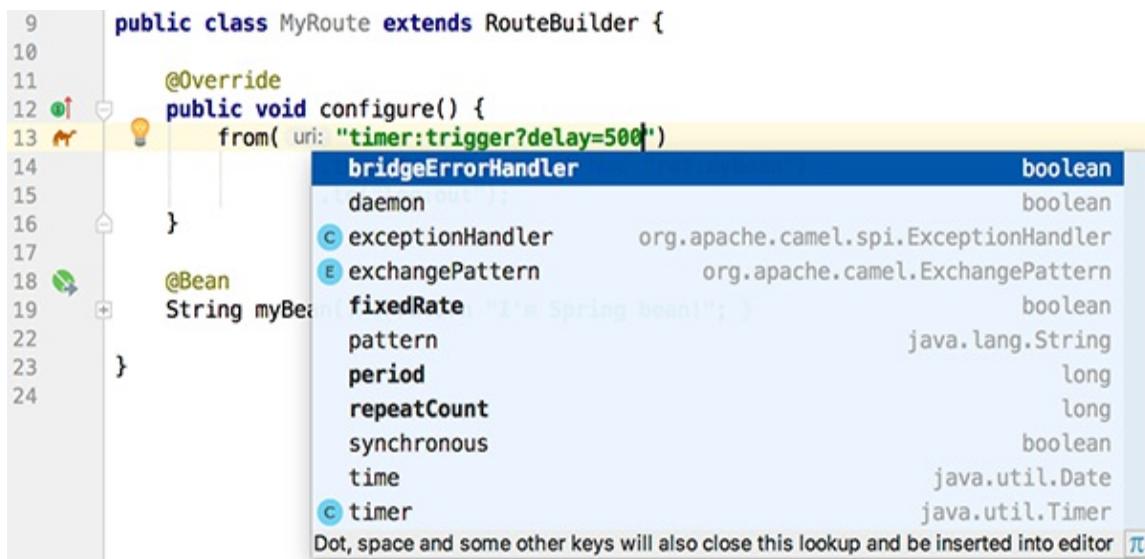
```
6  /***
7   * A simple Camel route
8   */
9  public class MyRoute extends RouteBuilder {
10
11     @Override
12     public void configure() {
13         from("timer:trigger?delay=500")
14             .transform().simple("text: ${ref:myBean}")
15             .to("log:out");
16     }
17
18     @Bean
19     String myBean() { return "I'm Spring bean!"; }
20
21
22 }
```

The screenshot shows the Java code for a Camel route named MyRoute.java. The code defines a class extending RouteBuilder and overrides the configure method. It uses a timer endpoint to trigger a route every 500 milliseconds, transforming the message using a Spring bean named myBean, and logging the result. The IDE's gutter on the left side of the code editor displays small Camel icons at the beginning of each Camel route detected in the source code. These icons include a green arrow pointing up, a yellow lightbulb, and a red person icon.

[Figure 19.7](#) In the gutter, there's a Camel icon at the beginning of every Camel route detected in the source code.

The plugin can, of course, do a lot more than show a little Camel icon. For example, position the cursor as shown in [figure 19.7](#), where the cursor is at the end of the timer endpoint. Then press Ctrl-space, which activates IDEA smart completion. Because the plugin is installed, it knows this is a Camel endpoint and therefore presents the user with a list of possible endpoint

options you can use to configure the endpoint, as shown in figure 19.8.



The screenshot shows a Java code editor with the following code:

```
9 public class MyRoute extends RouteBuilder {
10
11     @Override
12     public void configure() {
13         from("timer:trigger?delay=500")
14             bridgeErrorHandler
15             daemon
16             }
17
18     @Bean
19     String myBean
20     }
21
22
23
24 }
```

A code completion dropdown is open at the line `from("timer:trigger?delay=500")`, specifically at the `delay` parameter. The dropdown lists several options:

Option	Type
bridgeErrorHandler	boolean
daemon	boolean
c exceptionHandler	org.apache.camel.spi.ExceptionHandler
E exchangePattern	org.apache.camel.ExchangePattern
fixedRate	boolean
pattern	java.lang.String
period	long
repeatCount	long
synchronous	boolean
time	java.util.Date
timer	java.util.Timer

At the bottom of the dropdown, there is a note: "Dot, space and some other keys will also close this lookup and be inserted into editor".

Figure 19.8 Pressing Ctrl-space shows a list of possible options that can be used to configure the endpoint. The options highlighted in bold are the most commonly used options.

The list works as if you're doing smart completion on regular Java code. You can use *type ahead*, and by pressing Ctrl-J while the list is present, another window is shown on the side that presents documentation for the currently selected option in the list. This gives you information at your fingertips; you don't have to open a web browser and find the Camel component documentation. And if you want to go there, press Ctrl-F1, and IDEA will open the web page for the selected component for you in the browser.

You can also use Ctrl-space when configuring the values of the endpoint options. In figure 19.9, we're configuring the acknowledge mode option on a JMS endpoint.

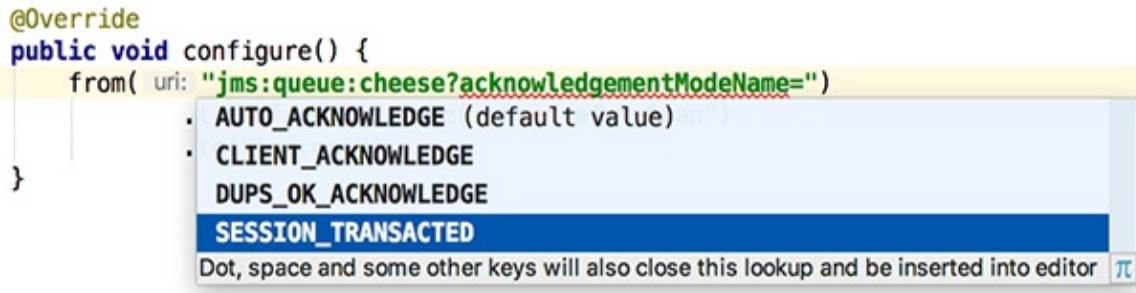


Figure 19.9 Press Ctrl-space while editing the value of an endpoint option to provide a list of possible choices if the option is an enum-based type, such as the JMS acknowledge-mode option.

The plugin has other noteworthy functionality such as live validation in the source code, where any misconfigured Camel endpoint or Simple language expression will be highlighted with a red underline.

If that's too much for you, the plugin can be configured from the preferences, where you find Apache Camel in the Languages & Frameworks group. The plugin works with both the Java and XML DSLs. Press Ctrl-space anywhere in your Camel code, and the plugin is there to help you. One last thing to highlight as well is that the plugin works with any Apache Camel release—it's capable of downloading the catalog information from newer or older versions on the fly (we cover the Camel Catalog in section 19.2).

A summary of the plugin's pros and cons is listed in table 19.2.

Table 19.2 The good and not so good about the Apache Camel IDEA tooling

Pros	Cons
Integrates natively with IDEA in the code editor	Works only on IDEA
Type-safe and real-time validation as you type in the code editor	No graphical visualization of Camel routes
Supports Java, and XML DSLs	No commercial support or paid work for engineers, meaning development effort is done in spare time
100% open source and Apache licensed	

Let's step outside IDEA and other IDE editors and explore a great little tool that comes out of the box with Apache Camel. It's the Apache Camel Maven plugin that's capable of scanning your source code and reporting invalid Camel endpoints.

19.1.3 CAMEL VALIDATION USING MAVEN

You may have experienced starting your Camel application and having it fail because one or more Camel endpoints were misconfigured due to an invalid option. It's both a blessing and a curse with Camel that endpoints are so quick and easy to configure using URI parameters. The endpoint URI is a string type, such as the following file endpoint:

```
file:src/data?noop=true&recursive=true
```

Now suppose you mistype the recursive option:

```
file:src/data?noop=true&recusive=true
```

Then when starting this application, Camel will report an error:

```
Caused by: org.apache.camel.ResolveEndpointFailedException:  
Failed  
to resolve endpoint: file://src/data?  
noop=true&recusive=true due to: There  
are 1 parameters that couldn't be set on the endpoint.  
Check the uri if the  
parameters are spelt correctly and that they are properties  
of the endpoint.  
Unknown parameters=[{recusive=true}]
```

Wouldn't it be great if you could validate those Camel URIs before you run your Camel application? Great news: the existing Camel Maven plugin from Apache Camel is capable of doing that.

**The story of the fabric8 Camel tooling
and the IDEA plugin**

This validation functionality that we're about to cover was first developed as an experiment at the fabric8 project (<https://fabric8.io>). After more than a year in active development at fabric8, the plugin code was stable and was donated to Apache Camel for inclusion from Camel 2.19 onward.

At the same time, the fabric8 team experimented with a Camel editor that was based on JBoss Forge and allows you to install the tooling as plugins to Eclipse, IDEA, and NetBeans. This tool was capable of type-safe editing of your Camel endpoints and EIP patterns in Java and XML DSL.

But we wanted tighter integration with the IDE editors than what JBoss Forge provides. On the evening of December 23, 2016, Claus created an experimental prototype of a Camel plugin for IDEA. This prototype was able to show a list of possible Camel options you could configure on the Camel endpoints. It provided type-safe code completions directly in the IDEA source code editor (as you have today for Java code). Over the next couple of months, this prototype matured into the Apache Camel IDEA plugin. The plugin code is hosted at GitHub at <https://github.com/camel-idea-plugin/camel-idea-plugin>.

The book's accompanying source code contains an example in the chapter19/camel-maven-validate directory. From this directory, you can run the following command from the command line:

```
mvn camel:validate
```

Running this command should report two errors:

```
[INFO] --- camel-maven-plugin:2.20.1:validate (default)
[INFO] Detected Camel version used in project: 2.20.1
[INFO] Validating using Camel version: 2.20.1
[WARNINg] Endpoint validation error at:
camelInaction.MyRouteBuilder.configure(MyRouteBuilder.java:
```

```
19)

    file:src/data?noop=true&recursive=true

        recursive      Unknown option. Did you mean:
[recursive]

[WARNING] Endpoint validation error at:
camelinaction.MyRouteBuilder.configure(MyRouteBuilder.java:
22)

log:uk?showall=true

        showall      Unknown option. Did you mean:
[showAll, showOut, showFiles]

[WARNING] Endpoint validation error: (2 = passed, 2 =
invalid,
  0 = incapable, 0 = unknown components)
[INFO] Simple validation success: (0 = passed, 0 = invalid)
[INFO] Duplicate route id validation success (0 = ids)
```

As you can see, the source code has two errors. The first is the mistype of the `recursive` option. The tooling gives you suggestions for the option. The second error is in a log endpoint, where we've configured the `showAll` option using a lowercase `a` in `all`. The correct option is spelled `showAll`.

ADDING THE PLUGIN TO YOUR PROJECT

To use the validation plugin in your Camel projects, add the following to the `pom.xml` file:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <version>2.20.1</version>
</plugin>
```

Then you can run the validation goal using the following:

```
mvn camel:validate
```

You can also enable the plugin to automatically run as part of the build, but configure the plugin to run the validate goal as part of the process-classes phase:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <version>2.20.1</version>
  <executions>
    <execution>
      <phase>process-classes</phase>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

By default, the plugin validates the source code only in src/main, so if you want to also validate the unit-test source code, you need to turn this on by configuring `includeTest=true`, as highlighted here in bold:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <version>2.20.1</version>
  <executions>
    <configuration>
      <b><includeTest>true</includeTest></b>
    </configuration>
    <execution>
      <phase>process-classes</phase>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

The plugin has options you can configure, and you can find details in the Camel plugin documentation at:
<https://github.com/apache/camel/blob/master/tooling/maven/camel-maven-plugin/src/main/docs/camel-maven-plugin.adoc>.

TIP The Camel Maven Plugin will include a new functionality (route-coverage goal) in Camel 2.21 that, after running your unit tests, allows you to generate route coverage reports. This can be used to identify whether you have tests that cover all parts of your Camel routes.

All the tools we've covered so far—JBoss Fuse Tooling, the IDEA Apache Camel plugin, and the Camel Maven plugin—use the Camel Catalog to obtain extensive information about what an Apache Camel release includes.

19.2 Camel Catalog: the information goldmine

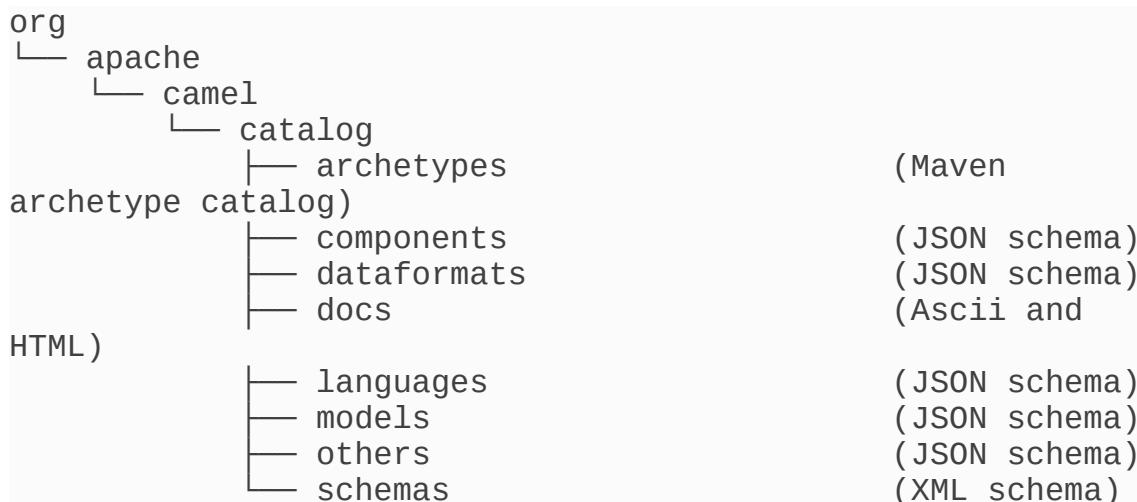
Starting from Camel version 2.17, each release includes a catalog with all sorts of information about what's included in the release. The catalog is shipped in an independent standalone camel-catalog JAR containing the following information:

1. List of all components, data formats, languages, EIPs, and everything else in the release
2. Curated lists for Spring Boot and Apache Karaf runtime
3. JSON schema with extensive details for every option
4. Human-readable documentation for every option
5. Categorization of options (for example, find all database components)
6. XML schema for the XML DSL
7. Maven Archetype Catalog of all the Camel archetypes
8. Entire website documentation as ASCII doc and HTML files
9. Validator for Camel endpoint and Simple language
10. API to create Camel endpoint URLs

1. Java, JMX, and REST API

The catalog provides a wealth of information that tooling can tap into and use. For example, JBoss Fuse Tooling uses the catalog in the graphical editor to know all details about every EIP and component Camel supports. The IDEA Camel tooling does the same thing. For example, the Maven validation plugin covered in section 19.1.3 uses the catalog during validation of all the Camel endpoints found while scanning the source code.

The camel-catalog JAR includes the information using the following directory layout:



Each directory contains files with the information. Every Camel component is included as JSON schema files in the components directory. For example, the Timer component is included in the file timer.json, as shown in the following listing.

Listing 19.1 JSON schema of the Timer component from camel-catalog

```
{  
  "component": {  
    "kind": "component",  
    "scheme": "timer", ①
```

①

The name of the component

```
"syntax": "timer:timerName", 2
```

2

URI syntax (without parameters)

```
"title": "Timer",
"description": "The timer component is used for
generating message
          exchanges when a timer fires.",
"label": "core,scheduling", 3
```

3

Labels are used for categorizing the component.

```
"deprecated": false,
"async": false,
"consumerOnly": true, 4
```

4

This component can be used only as a consumer (for example, from).

```
"firstVersion": "1.0.0", 5
```

5

First version when this component was included

```
"javaType":
"org.apache.camel.component.timer.TimerComponent",
"groupId": "org.apache.camel", 6
```

6

The Maven coordinate for the JAR that contains the component

```
"artifactId": "camel-core", 6
"version": "2.20.1" 6
},
"componentProperties": { 7
```

7

Component-level options

```
},  
"properties": { 8
```

8

Endpoint-level options (text abbreviated)

```
    "timerName": { "kind": "path", "group": "consumer",  
"required"  
        "bridgeErrorHandler": { "kind": "parameter", "group":  
"consume  
            "delay": { "kind": "parameter", "group": "consumer",  
"type": "  
                "fixedRate": { "kind": "parameter", "group":  
"consumer", "type":  
                    "period": { "kind": "parameter", "group": "consumer",  
"type":  
                        "repeatCount": { "kind": "parameter", "group":  
"consumer", "ty  
                            "exceptionHandler": { "kind": "parameter", "group":  
"consumer  
                                "daemon": { "kind": "parameter", "group": "advanced",  
"label":  
                                    "exchangePattern": { "kind": "parameter", "group":  
"advanced",  
                                        "pattern": { "kind": "parameter", "group": "advanced",  
"label"  
                                            "synchronous": { "kind": "parameter", "group":  
"advanced", "la  
                                                "time": { "kind": "parameter", "group": "advanced",  
"label": "  
                                                "timer": { "kind": "parameter", "group": "advanced",  
"label":  
                                            }  
                                        }  
                                    }
```

The JSON schema is divided into three parts:

- component—General information about the component
- componentProperties—Options you can configure on the component itself

- properties—Options you can configure on the endpoints

In the component section, you find the component name ❶ and syntax ❷. Then labels are used to categorize the component ❸. This component has the labels core and scheduling. In Camel, a component can be used as a consumer, producer, or both. The JSON schema indicates whether the component can be only a consumer or producer using consumerOnly ❹ or producerOnly, respectively. If neither of those entries exists, the component can be used as both. You can also see when a component was added to Apache Camel, as stated in the first version ❺ attribute. In this example, we can see the Timer component was included in the first release of Apache Camel. There's also information about which JAR file contains the component as Maven coordinates ❻. The last two parts document each option you can configure on the component level ❼ and endpoint level ⫽. Each entry in these parts is structured the same way. The Timer component has only endpoint-level options. Each option includes a lot of details that have been abbreviated in the listing. The following is a snippet of the delay option in its entirety:

```
"period": { "kind": "parameter", "displayName": "Period",
"group":
"consumer", "type": "integer", "javaType": "long",
"deprecated":
false, "secret": false, "defaultValue": 1000,
"description": "If
greater than 0 generate periodic events every period
milliseconds. The
default value is 1000. You can also specify time values
using units such as
60s (60 seconds) 5m30s (5 minutes and 30 seconds) and 1h (1
hour)."},
```

Let's break down the option and explain each detail, as listed in table 19.3.

Table 19.3 Breakdown of the component and endpoint options from the JSON schema

Option	Description

period	The name of the option.
kind	Kind of option, can either be path or parameter. A path is an option in the URI context path. Parameter denotes a URI query parameter.
displayName	The name of the option intended for displaying in UIs.
group	Group is a single overall name that categorizes the option.
type	JSON schema type of the option.
javaType	Java type of the option.
deprecated	Whether the option has been deprecated.
secret	Whether the option is sensitive, such as a username or password.
defaultValue	The default value of the option (optional).
description	Human-readable description.

The catalog also includes JSON schema files for every EIP, data format, and language in the Camel release. These schemas are structured in similar way as the components, as shown in [listing 19.1](#).

Subject of change

The Camel Catalog (`camel-catalog`) may include more details in future Camel releases. Therefore, the JSON schema may change and include additional information. But the basic structure as shown in [listing 19.1](#) is expected to stay.

If you want to write custom Camel tooling, the camel-catalog is a great source of information.

Let's move on to the web and look at Camel tooling with hawtio.

19.3 hawtio: a web console for Camel and Java applications

What is hawtio (<http://hawt.io>) and what does it do? Here's what hawtio says:

It's a pluggable management console for Java stuff that supports any kind of JVM, any kind of container (Tomcat, Jetty, Spring Boot, Karaf, JBoss, WildFly, fabric8, etc.), and any kind of Java technology and middleware.

—<http://hawt.io/faq/index.html>

If you've read this book in chronological order, you may remember that we talked about hawtio in previous chapters (such as chapter 16, where we briefly covered using hawtio to manage Camel applications). In this section, we'll take a deeper look at hawtio and unlock some of the features it provides that are relevant to Camel.

19.3.1 UNDERSTANDING HAWTIO FUNCTIONALITY

hawtio provides the following Camel features out of the box:

- Lists all running Camel applications in the JVM
- Details information of each Camel application such as version number and runtime statistics
- Lists all routes in each Camel application and their runtime statistics
- Manages the lifecycle of all Camel applications and their routes, so you can restart, stop, pause, or resume them

- Provides graphical representation of the running routes along with real-time metrics
- Provides embedded documentation of in-use components, endpoints, and EIPs
- Enables live tracing and debugging of running routes
- Profiles the running routes with real-time runtime statistics and details specified per processor
- Updates routes using a text-based XML editor (not persistent update)
- Enables browsing and sending messages to Camel endpoints

hawtio provides many Camel features out of the box. The most prominent Camel feature of hawtio is likely the Camel route diagram, shown in [figure 19.10](#).

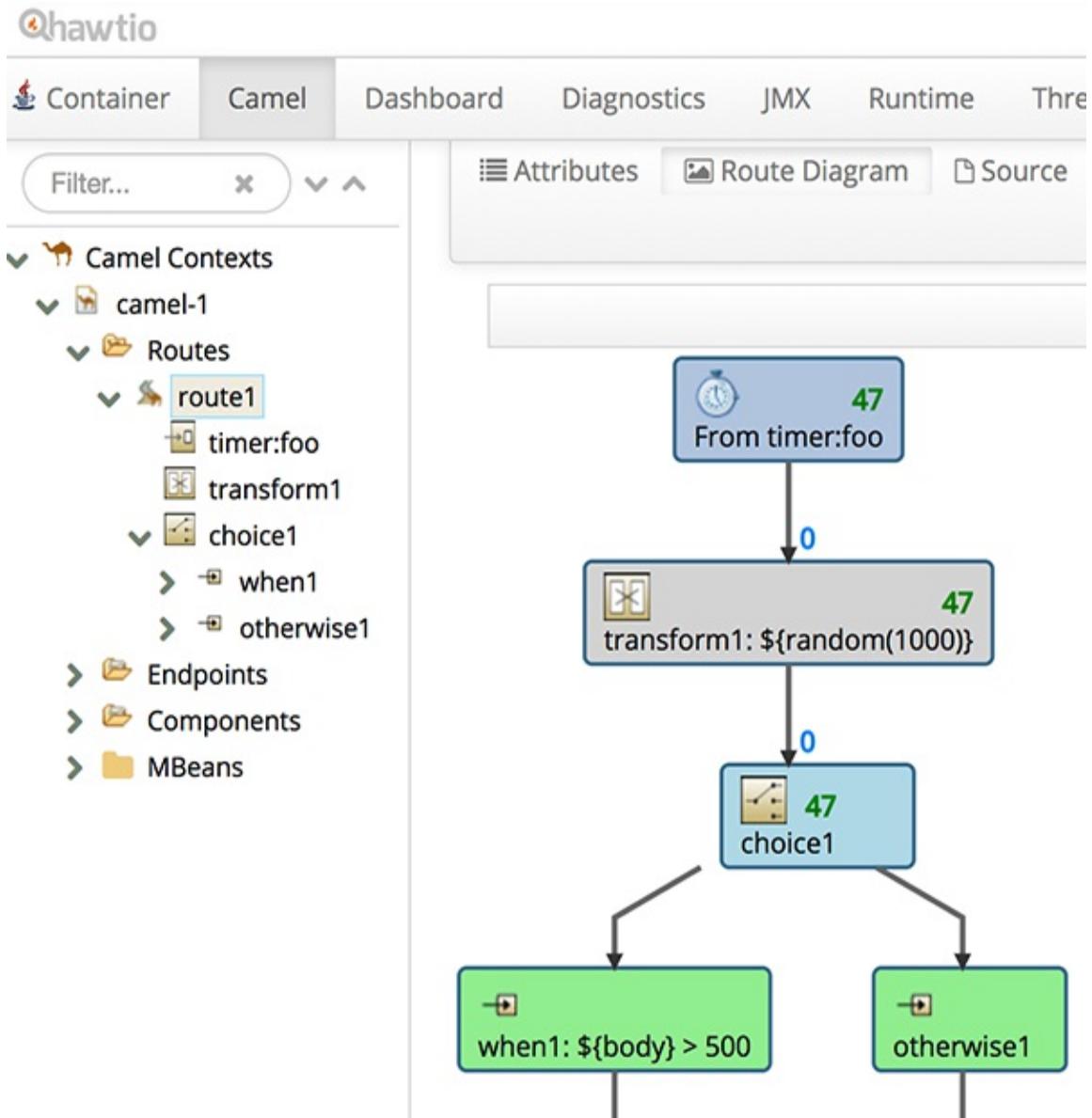


Figure 19.10 Visual representation of the running Camel routes using the hawtio web console. The numbers (47) are the messages processed, and the numbers (0) are the number of messages that are in flight (currently being processed). The numbers update in real time.

Installing hawtio in Karaf

You can install hawtio in Apache Karaf/ServiceMix using the following commands:

```
karaf@root()> feature:repo-add hawtio 1.5.6
```

```
Adding feature url mvn:io.hawt/hawtio-
karaf/1.5.6/xml/features
karaf@root()> feature:install hawtio
```

Then you can access hawtio in the web browser at <http://localhost:8181/hawtio> and log in with karaf/karaf (default username and password).

hawtio includes a route debugger, so let's try this in action.

19.3.2 DEBUGGING CAMEL ROUTES USING HAWTIO

When developing Camel routes, debugging at the route level can be handy. hawtio comes with a route debugger, which you can run from a web browser. Let's jump straight into action and debug the following route, which has been written in Java DSL:

```
from("timer:foo?period=5000")
    .transform(simple("${random(1000)}"))
    .choice()
        .when(simple("${body} > 500"))
            .log("High number ${body}")
            .to("mock:high")
        .otherwise()
            .log("Low number ${body}")
            .to("mock:low");

```

The route starts from a timer that triggers every fifth second. Then a random number between 0 and 1,000 is generated as the message body. Depending on whether the number is a high or low number, it's routed accordingly, using the Content-Based Router EIP. You can run this example, provided in the chapter19/hawtio-debug directory, from Maven using the following:

```
mvn compile exec:java
```

Because you want to debug the route, you can start the example with hawtio embedded by running the following Maven goal:

```
mvn compile hawtio:run
```

Then hawtio is started from the Maven plugin, which then starts the Camel application using a Java main class. The Java main class is configured in the Maven pom.xml file, as highlighted here:

```
<plugin>
  <groupId>io.hawt</groupId>
  <artifactId>hawtio-maven-plugin</artifactId>
  <version>1.5.6</version>
  <configuration>
    <mainClass>camelinaction.MainApp</mainClass>
  </configuration>
</plugin>
```

When you run the `hawtio:run` goal, hawtio and the Camel application are started together in the same JVM. And after a little while, hawtio opens automatically in your web browser. You can then use the web browser to see what happens at runtime in the JVM where Camel is running. For example, you can see real-time metrics of the number of messages being processed.

Okay, let's get on with it. How do you debug a Camel route using hawtio? You need to select the route you want to debug in the tree on the left side, and because there's only one route in this example, you should select `route1`. In the center of the screen, click the Debug button and then click the Start Debugging button, which should show the route. You can then double-click an EIP to add a breakpoint—for example, on the `choice1` EIP. After a little while, the EIP should display a blue ball, which indicates that a breakpoint was hit. At the bottom of the screen, the message is listed, and you can click it to show its message body and headers. After doing this, you should have a screen similar to [figure 19.11](#).

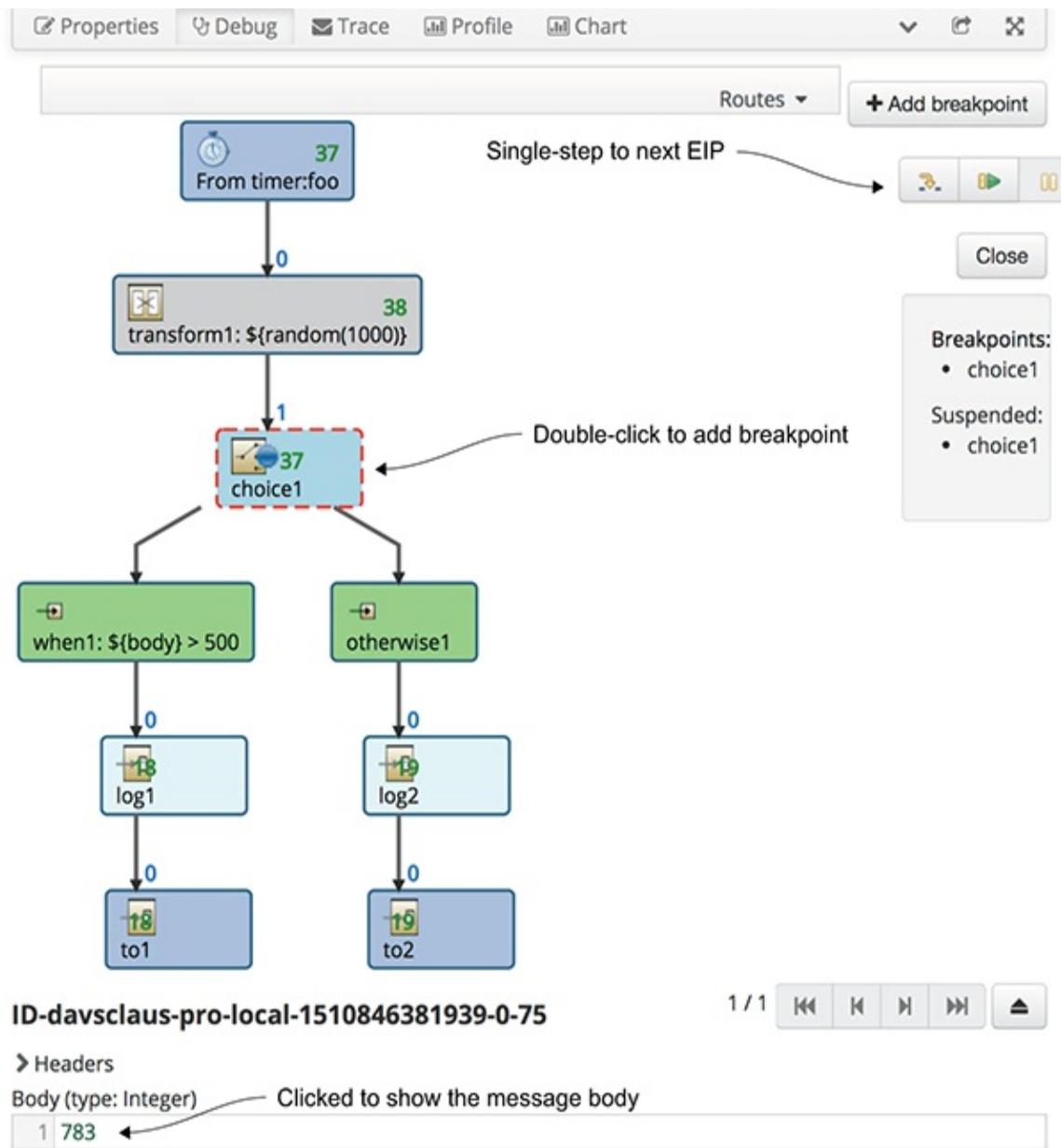


Figure 19.11 Using hawtio to debug a Camel route. Double-click an EIP to set a breakpoint. When a breakpoint is hit, the ball color changes to blue, and at the bottom of the screen you can see the current message body and its headers. At the top right, you can click the Single Step button to continue routing and pause the debugger at the next EIP.

In figure 19.11, the debugger has paused at the Content-Based Router EIP (marked *Choice*), which is indicated by the blue ball. At the bottom of the screen, you can see that the message body has the value of 783 (you need to click the message to expand it and show the content of the message). When you click the Single Step button at the top-right corner, the message is routed and

paused at the next EIP, which is the branch on the left-hand side in the Content-Based Router (marked *When*) because 783 is higher than 500. When you're finished with the debugger, remember to click the Close button.

Debugging API

The debugger in hawtio and the Eclipse-based JBoss Fuse Tooling both use the same debugging API from camel-core. This API is available to custom tooling from JMX from the `ManagedBacklogDebugger` MBean, which you can find under the tracer tree in the Camel JMX tree. This MBean has a rich set of debugging operations to control breakpoints, run in single-step mode, update the message body/headers, and more.

You may not always be able to run your Camel application with hawtio embedded. Therefore, you can run hawtio in another JVM and connect from hawtio to your existing Camel applications running in another JVM.

CONNECTING HAWTIO TO EXISTING RUNNING JVM

If you have any existing Java application running in a JVM, you can start hawtio in standalone mode and then from hawtio connect to the existing JVM. In the example located in the chapter19/hawtio-directory, instead of running the `hawtio:run` plugin, you run `exec:java`:

```
mvn compile exec:java
```

Then from another terminal, you download hawtio-app and run it as a standalone Java application:

```
java -jar hawtio-app-1.5.6.jar
```

When hawtio opens in the web browser, the Connect plugin

should be selected by default. In the center of the screen, you click the Local button, which lists all the local JVMs running on your computer. One of the rows in the table should have the text exec:java, which is the JVM that runs the Camel application. You can then click the Play button on the right-hand side to create a connection to this JVM, which then shows a hyperlink. After you've done this, you should have a screen similar to [figure 19.12](#).

PID	Name	Agent URL
71477	org.codehaus.plexus.classworlds.launcher.Launcher exec:java	http://127.0.0.1:56737/jolokia/
71782	IntelliJ IDEA	
71771	hawtio	
71710		

Figure 19.12 Running hawtio in standalone mode, to connect to any local JVMs running on your computer. When you click the hyperlink in the Agent URL, hawtio opens a web page connected to that JVM. This allows you to manage and look inside that running JVM (for example, the Camel plugin if Camel is running).

If you're interested in building web tooling for Java applications, we encourage you to look into hawtio and the technologies it's using, such as AngularJS, Bootstrap, TypeScript, and Jolokia. You can find more hawtio plugin examples from the hawtio website (<http://hawt.io>) and from its source code hosted on GitHub (<https://github.com/hawtio/hawtio>).

19.4 Summary and best practices

In this chapter, we've shown you the best Camel tools you can find on the internet. The best-known tools are the Camel editors for graphical drag-and-drop development, such as the Eclipse tooling from JBoss. But a set of smaller and lighter tools is on the rise, such as the Apache Camel IDEA plugin, which focuses on

extending the source code editor instead of on graphical drag-and-drop development. It's expected that some of these capabilities will find their way into Eclipse as well.

You learned that all the Camel components, data formats, EIPs, languages, and everything else are all cataloged in the Camel Catalog, which allows tools to know *everything*. The tools from JBoss and the Camel IDEA plugin all use this to provide type-safe editors for editing Camel endpoints and EIPs.

You also shouldn't miss out on the Camel Maven plugin that can check the source code during compile time to catch misconfigured Camel endpoints. Many Camel users have found great value with the hawtio web console, so make sure to try it out as well.

The takeaways from this chapter are as follows:

- *Graphical tooling*—New Camel developers and less experienced Java developers may find value in starting their Camel development by using the graphical editor from JBoss in Eclipse.
- *Camel route debugger*—A powerful feature from the JBoss Eclipse tooling is the Camel route debugger, which is integrated seamlessly into Eclipse. As an alternative debugger, you can use hawtio from a web browser.
- *Type-safe endpoint editor*—The Camel IDEA tooling provides a lightweight editor that allows you to edit/add endpoints directly in the source code from the cursor position. We think this kind of tooling will appeal to both new and experienced Camel developers.
- *Camel Maven plugin*—Don't miss out on the Maven plugin that you can run during compilation to check for misconfigured Camel routes.
- *Use the Camel Catalog*—The Camel Catalog is a goldmine of information that tooling developers should tap into. All the tools covered in this chapter are using the catalog.

With the goldmine of information from the Camel Catalog and

the powers from Jolokia, Maven, IDEA, Eclipse, and hawtio, we think you have powerful ways to build custom tools you may need for your Camel-based integration applications.

These tools are all open source and have an open community. We welcome you to participate and help improve these tools with features you may find useful, or to have some fun with hacking on code that isn't your typical day-to-day job.

GOODBYE FROM CLAUS AND JONATHAN

With that, we've come to the end of our not-so-little book on Apache Camel. In this second edition, we tried to cover as much of Camel's core features as possible, with the addition of how to use Camel in more modern settings, such as a framework for microservices, deployed in the cloud with Docker and Kubernetes, or even as a reactive engine, as we discuss in the bonus chapter (available online only). Believe it or not, there's still a lot more material. For that, though, we refer you to the Camel website, which has a full reference of all 200+ components. Besides, reading (or writing) a book about all 200 components wouldn't be the most exciting thing in the world.

One final note about Camel is that it is an active project and changes quickly. We suggest keeping up with Camel via its community (discussed in appendix B) and staying current on new features. Maybe you'll even find the inspiration to contribute to Apache Camel yourself.

We hope you've enjoyed the *Camel in Action*, 2nd Edition ride. We surely enjoyed making it yet again to the finish line after two and a half years in the making. Hope to see you around in the Camel community—as we say many times, we love contributions!

PS: If you like Apache Camel, Claus and Jonathan Cheers,
we'd appreciate it if you'd give Camel a star on GitHub
(<https://github.com/apache/camel>), and you're also quite
welcome to star this book
(<https://github.com/camelinaction/camelinaction2>) as well.

20

Reactive Camel

This bonus online chapters covers

- First steps with Reactive Streams
- Using Reactive Streams with Camel
- Using Eclipse Vert.x with Camel

If you read this book in chronological order, you've been on a long journey. This is the first of two bonus online-only chapters from the hands of Claus and Jonathan, and this time we'll keep it short. We have only two topics we want to bring to your attention here at the end.

Apache Camel is a well-established project that's been around for over a decade. A decade in the IT industry is like half a lifetime for humans. Only recently has reactive programming started to gain more interest, especially since Java 8 added support for `java.util.stream` in its streaming API. Further interest may be spurred with the upcoming Spring Framework 5, which now includes a reactive engine.

As for Apache Camel, the current architecture of Camel 2.x is based on a hybrid routing engine that executes both blocking and nonblocking processing, depending on which EIPs and components are being used. The upcoming Camel 3.x architecture is intended to be a dual engine comprising the current hybrid engine and a new reactive engine based on a

reactive event bus.

The first half of this chapter presents a brief look at the new camel-reactive-streams component that's introduced in Camel 2.20. This component allows users to take advantage of the reactive streaming APIs and work with the many Camel components. The second half of this chapter is a personal story from Claus, who built a Vert.x application that can simulate live football scores.

20.1 Using Reactive Streams with Camel

This section covers the new camel-reactive-streams component, which you can use to integrate Camel with the Reactive Streams API (www.reactive-streams.org). This API is a small standard specification for asynchronous stream processing. Java 9 comes with the Flow API (`java.util.concurrent.flow`), which corresponds to the Reactive Streams API.

20.1.1 REACTIVE STREAMS API

The API is composed of the following four interfaces:

- Publisher—A Publisher is a provider of a potentially unbounded number of sequenced elements, publishing them according to the demand received from its subscribers.
- Subscriber—A Subscriber receives events from a Publisher.
- Processor—A Processor represents a processing stage, which is both a Subscriber and a Publisher and obeys the contracts of both.
- Subscription—A Subscription represents a one-to-one lifecycle of a Subscriber subscribing to a Publisher.

Because Reactive Streams is a specification, you need to use a library that implements this specification. RxJava and Reactor Core are two widely used reactive programming libraries that use Reactive Streams with back pressure included. This book

presents examples using those two libraries.

The Reactive Streams specification defines a model for *back pressure*, a way to ensure that a fast publisher doesn't overwhelm a slow subscriber. Back pressure provides resilience by ensuring that all participants in a stream-based system participate in flow control to ensure a steady state of operation and graceful degradation.

20.1.2 REACTIVE FLOW CONTROL WITH BACK PRESSURE

In an ideal situation, a publisher is able to push data to a subscriber as fast as possible, and the subscriber is able to keep up. This isn't the case in the real world we live in, and it's easy to imagine situations in which the subscriber can't keep up and becomes flooded with data. You could try to deal with this by having a buffer at the consumer side with a reasonable capacity to store new data while the consumer is busy processing. But this often mitigates only *small bumps in the road*. If the situation continues and the producer remains faster than the consumer, the buffer will eventually run out of capacity.

Okay, you could then choose a strategy to start dropping data if the buffer is full. For example, you could choose to drop the oldest or the newest data. But that will result in data loss, which often isn't desirable.

What you need is a bidirectional flow of data. Data flows downstream from the publisher to the subscriber, and the subscriber sends a signal upstream to demand more data. Figure 20.1 illustrates this principle.

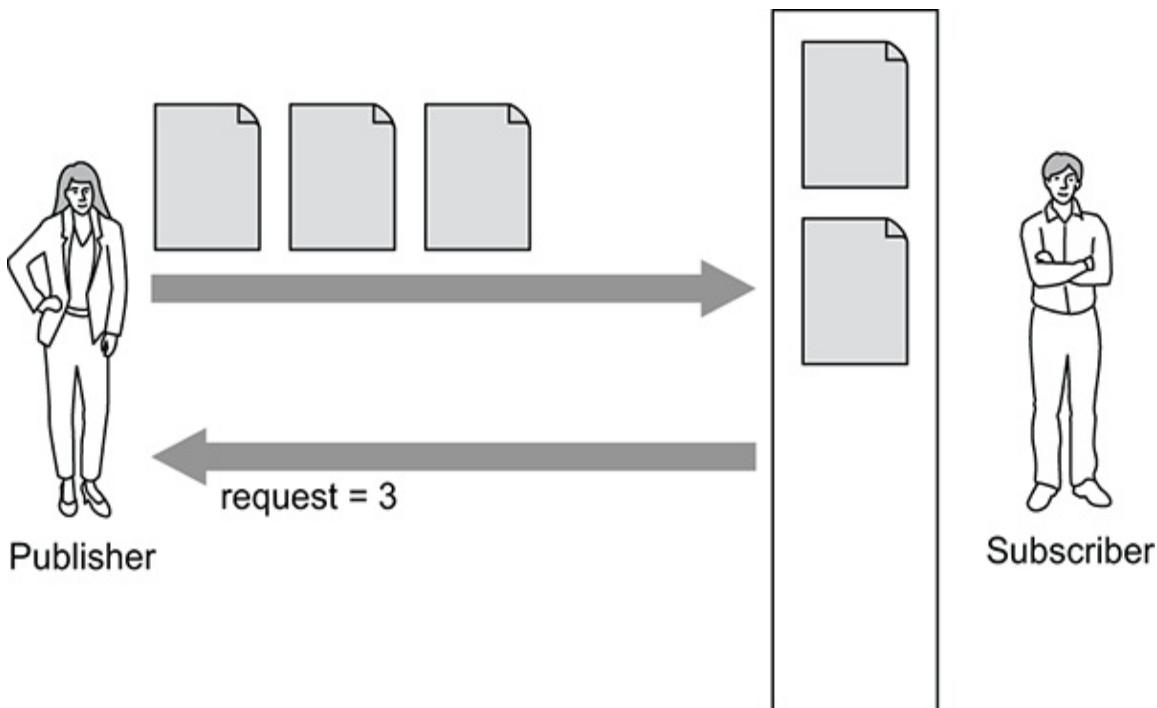


Figure 20.1 The Publisher Sends data to the Subscriber, and the Subscriber requests more data from the Publisher. Both events occur asynchronously.

When the publisher receives a demand from the subscriber, it's free to publish new data up to the number of elements requested. The bidirectional flow between the publisher and the subscriber is asynchronous and guarantees the best possible flow control.

The back pressure doesn't terminate at the first publisher, as it may cascade further up to upstream publishers. This follows one of the topics from the Reactive Manifesto:

Back-pressure may cascade all the way up to the user, at which point responsiveness may degrade, but this mechanism will ensure that the system is resilient under load, and will provide information that may allow the system itself to apply other resources to help distribute the load.

—www.reactivemanifesto.org/glossary#Back-Pressure

To better understand how the flow model between the Publisher and subscriber works, look at [figure 20.2](#). The most interesting aspect in [figure 20.2](#) is that the Subscriber can request more data

using the `request(limit)` method. Then the Publisher sends data to the Subscriber up until that limit. At any time, the Subscriber can request more data by calling the `request(limit)` method again.

You should also know that a Subscriber should subscribe to only one Publisher, whereas a Publisher can have one or more Subscribers.

Okay, enough talk; let's get down to action.

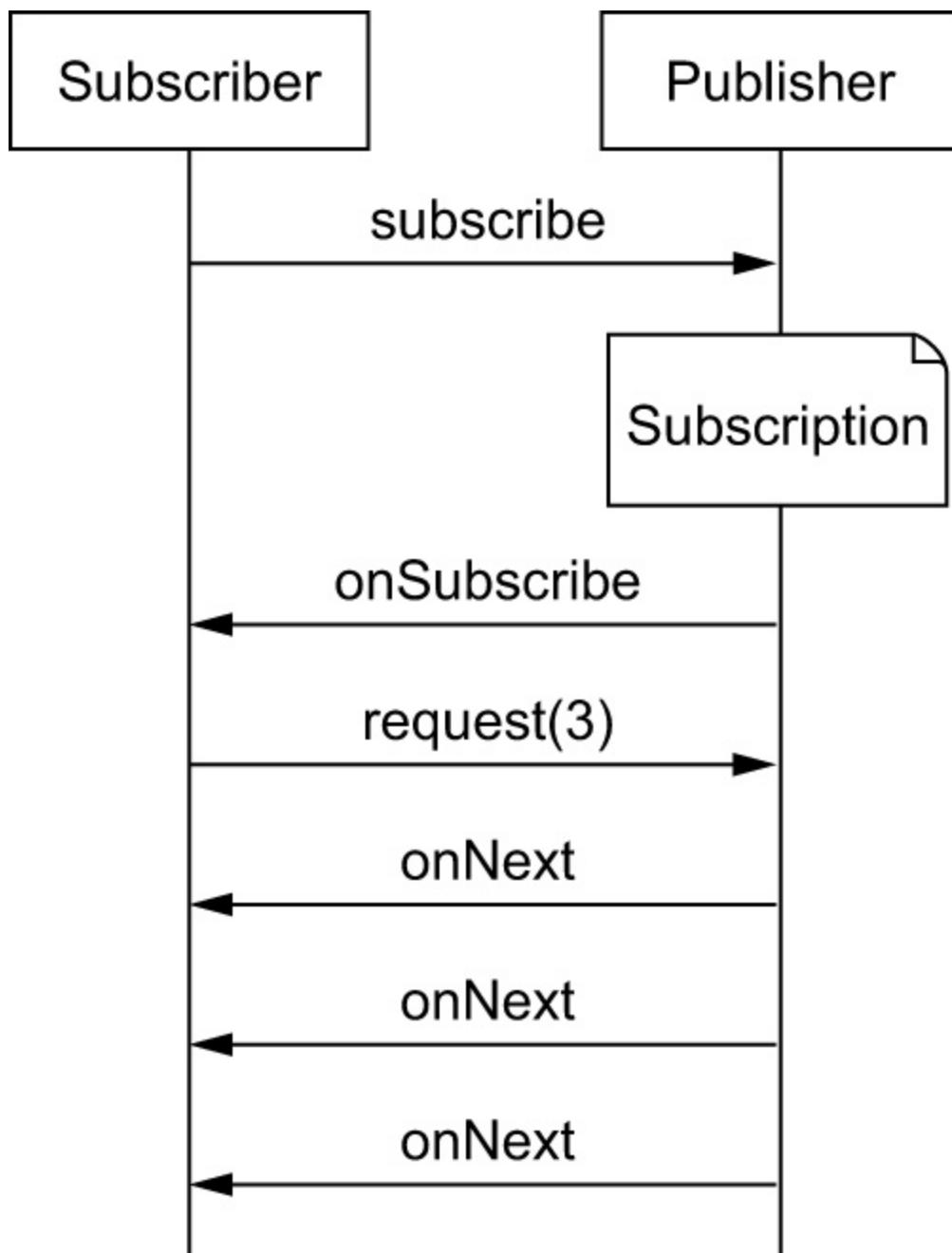


Figure 20.2 The Subscriber subscribes to a Publisher, which creates a Subscription. The Subscriber will be signaled by the `onSubscribe` callback when it has been successfully subscribed and the Producer is ready to send data. The Subscriber then requests how much data to receive. The Subscriber receives each data message per the `onNext` callback up until the number of elements requested.

20.1.3 FIRST STEPS WITH REACTIVE STREAMS

To understand the gist of building reactive flows, we'll start with

a simple example: we'll take a set of words, apply a function, and then log each word. To make the example as simple as possible, we'll use a fixed set of words (not a continued stream), and the reactive engine supports this by making it possible to create a Publisher that can do only that. The following listing shows how this can be coded using RxJava as the reactive engine.

Listing 20.1 Reactive flow takes the words that are uppercased and logged

```
Publisher<String> publisher = Flowable.just("Camel",
"rocks",
"streams",
"as", "well"); 1
```

1

Publisher with only these words

```
Subscriber<String> subscriber = new DefaultSubscriber<>()
{ 2
```

2

Creates Subscriber

```
    public void onNext(String w) { 3
```

3

The callback executed for each new item

```
        LOG.info(w.toUpperCase());
    }  
}
```

```
    public void onError(Throwable throwable) 4
```

4

The callback executed when an error is encountered

```
    }

    public void onComplete() { ⑤
```

⑤

The callback executed when the stream ends

```
}  
}  
  
publisher.subscribe(subscriber);
```

The main entry to using Reactive Streams with RxJava is the `io.reactivex.Flowable` class ①, where you can create a Publisher to stream from an array. The Subscriber is created from the `io.reactivex.subscribers.DefaultSubscriber` class ② that allows you to define three callbacks (methods) to receive items and notifications from the Publisher. The `onNext` callback ③ will be called once per every item of the stream. The `onError` callback ④ will be called if an error occurs upstream. The `onComplete` callback ⑤ will be called when the stream completes normally (for bounded streams, like the one used in this example), to signal that no more items will be pushed downstream. In case of error, only the `onError` callback will be called, and `onComplete` will be skipped.

In RxJava, subscribers shouldn't be created explicitly using the `DefaultSubscriber` class. Instead, callback functions and transformations are usually attached to a stream using a flow DSL, as shown in the following listing.

Listing 20.2 Reactive flow using flow DSL

```
Publisher<String> publisher = Flowable.just("Camel",
"rocks",
"streams", "as",
"well"); ①
```

①

Publisher with only these words

```
Flowable.fromPublisher(publisher) ②
```

②

Defines a flow starting from the Publisher

```
.map(w -> w.toUpperCase()) ③
```

③

Applies uppercase function

```
.doOnNext(w -> LOG.info(w)) ④
```

④

onNext logs the word

```
.subscribe(); ⑤
```

⑤

Subscribes

This Publisher is created the same way as in [listing 20.1](#), from Flowable ① to publish just the given words from the array. The Subscriber is created using Flowable ②, where you specify the Publisher to be used. You then want to uppercase each word, which is done via the map function, where you can use Java 8 lambda style to call the toUpperCase method ③. The flow then continues, where you want to run some code that can log the word, as done in the doOnNext method ④. And finally, you call the subscribe method ⑤, and the flow gets going.

When defining Reactive Streams using this kind of flow style, it may be more common to set up the entire flow directly on one Flowable. The code from [listing 20.2](#) can be compacted to four lines of code:

```
Flowable.just("Camel", "rocks", "streams", "as", "well")
    .map(String::toUpperCase)
    .doOnNext(LOG::info)
    .subscribe();
```

You can find this example with the source code in chapter20/rx-java2, and you can try the example using Maven:

```
mvn test -Dtest=FirstTest
```

You can also find an equivalent example using the Reactor Core engine instead in the chapter20/reactor-core directory, and you can try that example with the following:

```
mvn test -Dtest=FirstTest
```

The two examples are almost identical. When using Reactor Core, you use reactor.core.publisher.Flux instead of io.reactivex.Flowable.

This was a good first test, but let's move on to see how you can use Camel together with Reactive Streams.

20.1.4 USING CAMEL WITH REACTIVE STREAMS

The reactive-streams component is a Camel component that allows you to use Camel as a publisher or subscriber with your Reactive Streams. In this section, you'll add Camel to the previous example and then let Camel act as a publisher to send in the words to the Reactive Streams. The following listing shows how this can be done.

[Listing 20.3](#) Using Camel as a Publisher to send in words to the reactive flow

```
CamelContext camel = new DefaultCamelContext();
CamelReactiveStreamsService rsCamel =
CamelReactiveStreams.get(camel); ①
```

①

Creates Reactive Camel

```
camel.start(); 2
```

2

Starts Camel

```
Publisher<String> publisher = rsCamel.from("seda:words",  
String.class); 3
```

3

Creates Camel Publisher to seda:words endpoint

```
Flowable.fromPublisher(publisher) 4
```

4

Reactive flow

```
.map(w -> w.toUpperCase()) 4  
.doOnNext(w -> LOG.info(w)) 4  
.subscribe(); 4
```

```
FluentProducerTemplate template =  
camel.createFluentProducerTemplate(); 5
```

5

Sends data using Camel

```
template.withBody("Camel").to("seda:words").send(); 5  
template.withBody("rocks").to("seda:words").send(); 5  
template.withBody("streams").to("seda:words").send(); 5  
template.withBody("as").to("seda:words").send(); 5  
template.withBody("well").to("seda:words").send(); 5
```

To bridge Camel with Reactive Streams, you should get an instance of `CamelReactiveStreamsService` **1**, whose lifecycle is controlled by the `camelContext` **2**. Camel can be used as Publisher from any of its 200+ components that can act as a

consumer. At first, it may sound confusing that a Reactive Streams Publisher is using a Camel consumer. The Publisher is the data sink where new data comes in, which is a consumer in EIP terms and hence a Camel consumer.

In this example, you want to receive data from the `seda:words` Camel endpoint **③**. Notice that you specify the type as `String`, which ensures that the `Flowable` builder **④** is able to use this as a Java generic, so that the compiler can accept the lambda code that calls the `toUpperCase` method, because it knows it's a `String` type. The last part of the example is to use Camel to send the words to the `seda:words` endpoint **⑤**, which then triggers the reactive flow.

You can find this example with the source code in `chapter20/rx-java2`, and you can try the example using this Maven goal:

```
mvn test -Dtest=CamelFirstTest
```

USING CAMEL ROUTE AS REACTIVE STREAMS PUBLISHER

When using Camel, you often use Camel routes, so let's take a look at how to let a Camel route be the data sink for a reactive flow. This time, let's try numbers instead of words and use a Camel route that has a continued stream of data, as shown here:

```
from("timer:number")
    .transform(simple("${random(0,10)}"))
    .log("Generated random number ${body}")
    .to("reactive-streams:numbers"); ①
```

①

Sends data to reactive-streams endpoint

The Camel route is trivial, which starts from a timer that triggers once per second. You then generate a random number between 0 and 9 (inclusive), which is sent to the `reactive-streams` endpoint with the name `numbers` **①**. This is a Camel endpoint from the `camel-reactive-streams` component, which you can use as bridge

between Camel and your reactive flows. The following listing shows the source code with the reactive flow.

Listing 20.4 Reactive flow with a Publisher as data sink from Camel route

```
CamelReactiveStreamsService rsCamel =  
CamelReactiveStreams.get(context); 1
```

1

Creates Reactive Camel

```
Publisher<Integer> numbers =  
rsCamel.fromStream("numbers", 2)
```

2

Publisher from Camel reactive-streams endpoint

```
Integer.class); 2
```

```
Flowable.fromPublisher(numbers) 3
```

3

Reactive flow

```
.filter(n -> n > 5) 4
```

4

Filters for only big numbers

```
.doOnNext(n -> log.info("Streaming big number {}",  
n)) 5
```

5

Logs the big number

```
.subscribe();
```

⑥

⑥

Subscribes

To use Camel with reactive flows, you need to get an instance of `CamelReactiveStreamsService` ①, which you use to create a `Publisher` from the stream with the name `numbers` of type `Integer` ②. Pay attention that `numbers` is the same name used in the Camel route where data is being sent by the following:

```
.to("reactive-streams:numbers")
```

The reactive flow starts from the `Publisher` you created ③. This time, you want to apply a filter that drops the low numbers so you carry only the big numbers ④. Each of these big numbers is then logged ⑤. Finally, you start this flow by calling the `subscribe` method ⑥.

Reactive Streams operators

When using Reactive Streams with RxJava or Reactor Core, you have many operators at your disposal to apply all kind of functions to filter, transform, combine, and merge streams. In this chapter, you've used only a few of those operators. The ReactiveX website lists all the operators: <http://reactivex.io/documentation/operators.html>.

You can try this example, provided in the `chapter20/rx-java2` directory, with the following:

```
mvn test -Dtest=CamelNumbersTest
```

USING CAMEL ROUTE AS REACTIVE STREAMS SUBSCRIBER

Now let's try the opposite: letting a reactive flow publish streams with Camel acting as the subscriber by receiving the data as

input to a Camel route. The Camel route is merely two lines of code:

```
from("reactive-streams:numbers")
    .log("Got number ${body}");
```

The reactive flow source code is shown in the following listing.

Listing 20.5 Reactive flow with Camel as a Subscriber

```
CamelReactiveStreamsService rsCamel =
CamelReactiveStreams.get(context); 1
```

1

Creates Reactive Camel

```
Flowable.just("3", "4", "1", "5", "2") 2
```

2

Publisher that just sends those five numbers

```
.sorted(String::compareToIgnoreCase) 3
```

3

Sorts the numbers

```
.subscribe(rsCamel.streamSubscriber("numbers",
String.class)); 4
```

4

Subscriber from Camel reactive-streams endpoint

Yes, you've read it before: when you use Camel with Reactive Streams, you need to get an instance of CamelReactiveStreamsService **1**. To keep this example simple, you let RxJava create a Publisher with just five numbers **2**. Because the numbers are unordered, you can apply a sort

function ③. Then you create a Camel Subscriber with the name numbers ④. The name of the stream is the name of the endpoint used in the Camel route, for example:

```
from("reactive-streams:numbers")
```

You can try this example from the chapter20/rx-java2 directory by executing the following Maven goal:

```
mvn test -Dtest=CamelConsumeNumbersTest
```

So far, all the integration between Camel routes and reactive flows has used the reactive-streams component. But regular Camel endpoints can also be used.

USING REGULAR CAMEL COMPONENTS IN REACTIVE FLOW

The last example we want to show you uses regular Camel endpoints in the reactive flow. In this example, you'll use the Camel file component as a sink for a reactive flow and a Camel route as the subscriber. The following listing shows the source code.

Listing 20.6 Reactive flow with regular Camel endpoints and routes

```
Flowable.fromPublisher(rsCamel.from("file:target/inbox"))
```

1

Publisher from Camel file endpoint

```
.doOnNext(e -> rsCamel.to("direct:inbox", e))
```

2

Calls Camel route from flow

```
.filter(e ->  
e.getIn().getBody(String.class).contains("Camel"))
```

3

3

Filters out files without Camel text

```
.subscribe(rsCamel.subscriber("direct:camel"));
```

4

Subscriber from Camel direct endpoint

```
from("direct:inbox")
```

5

Camel route called from flow

```
.log("Inbox ${header.CamelFileName}")  
.wireTap("mock:inbox");
```

```
from("direct:camel")
```

6

Camel route used by Subscriber

```
.log("This is a Camel file ${header.name}")  
.to("mock:camel");
```

This time, the reactive flow seems more complicated the first couple of times you read it. You start using reactive Camel to create a Publisher from a regular Camel file endpoint ①. From the reactive flow, you can make calls into Camel routes from within the `doOnNext` function ②, where you can use Reactive Camel to publish to the Camel route ⑤. Then you use the `filter` function ③ to include only files that contain the text “Camel”. Finally, you let Camel be the Subscriber ④ by routing the data to the specified Camel route ⑥.

You can try this example, found in the `chapter20/rx-java2` directory, by running the following:

```
mvn test -Dtest=CamelFilesTest
```

TIP You can find more examples from Apache Camel at <https://github.com/apache/camel/tree/master/examples/camel-example-reactive-streams>.

Section 20.1.1 mentioned that one of the main goals of the Reactive Streams specification is to define a model for back pressure. The following two sections show you how to configure and control back pressure from a Camel reactive producer and consumer point of view.

20.1.5 CONTROLLING BACK PRESSURE FROM THE PRODUCER SIDE

When routing messages using Camel to an external subscriber, back pressure is by default handled by an internal buffer that caches the Camel messages before delivering them to the reactive subscriber. If the subscriber is slower than the rate of messages, the internal buffer may fill up and become too big. Such a situation must be avoided.

For example, suppose you have the following Camel route:

```
from("jms:queue:inbox")
    .to("reactive-streams:inbox");
```

If the JMS queue contains a lot of messages and the reactive subscriber is too slow to process the messages, the pending messages will keep piling up in an internal buffer by the reactive-streams endpoint. That can potentially degrade the performance in the JVM by taking up memory, or in the worst case, cause an out-of-memory error in the JVM and cause the application to crash.

We've provided an example, shown in the following listing, with the source code demonstrating a situation in which the subscriber is too slow, and pending messages are piling up in the internal buffer.

Listing 20.7 Reactive flow without back pressure causing

messages to fill up buffer

```
public void testNoBackPressure() throws Exception {  
    CamelReactiveStreamsService rsCamel =  
    CamelReactiveStreams.get(context);  
  
    Publisher<String> inbox = rsCamel.fromStream("inbox",  
String.class);  
  
    Flowable.fromPublisher(inbox)  
        .doOnNext(c -> {  
            log.info("Processing message {}", c); 1  
        })  
        .subscribe();  
}
```

1

Reactive flow that simulates a slow subscriber

```
    Thread.sleep(1000); 1  
}  
.subscribe();  
  
for (int i = 0; i < 200; i++) { 2
```

2

Sends in 200 messages to queue

```
    fluentTemplate.withBody("Hello " + i) 2  
        .to("seda:inbox")  
waitForTaskToComplete=Never).send(); 2  
}  
  
Thread.sleep(250 * 1000L);  
}  
  
protected RoutesBuilder createRouteBuilder() throws  
Exception {  
    return new RouteBuilder() {  
        public void configure() throws Exception {  
            from("seda:inbox")  
                .delay(100) 3
```

3

Camel route being faster than the subscriber

```
.log("Camel routing to Reactive Streams: ${body}")  
.to("reactive-streams:inbox"); 4
```

4

Messages will pile up here at the reactive buffer

```
}  
};  
}
```

This example has been constructed so that we humans can follow what happens while it runs; the reactive flow takes 1 second to process each message **1**. Instead of sending in hundreds of thousands of messages, you use a low limit of 200. That's enough to prove the point that those messages will stack up at the reactive buffer **4**. The Camel route also has a little delay **3** to allow us to better follow from the console output what's happening.

You can run the example from the chapter20/rx-java2 directory by executing the following Maven goal:

```
mvn test -Dtest=CamelNoBackPressureTest
```

While the example runs, it outputs to the console, as shown here:

```
INFO NoBackPressureTest - Processing message Hello 20  
INFO route1 - Camel routing to Reactive  
Streams: Hello 195  
INFO route1 - Camel routing to Reactive  
Streams: Hello 196  
INFO route1 - Camel routing to Reactive  
Streams: Hello 197  
INFO route1 - Camel routing to Reactive  
Streams: Hello 198  
INFO route1 - Camel routing to Reactive  
Streams: Hello 199  
INFO NoBackPressureTest - Processing message Hello 21  
INFO NoBackPressureTest - Processing message Hello 22
```

The interesting part of the output is highlighted in bold. Here you can see that Camel has routed all 200 (0–199) messages to the Reactive Streams channel. And at this time, the reactive flow

is processing message number 21, which means that $200 - 21 - 1 = 178$ messages are pending in the buffer. Now suppose you send 2,000 messages instead of 200; what would the situation be?

```
INFO route1          - Camel routing to Reactive  
Streams: Hello 1998  
INFO route1          - Camel routing to Reactive  
Streams: Hello 1999  
INFO NoBackPressureTest - Processing message Hello 205  
INFO NoBackPressureTest - Processing message Hello 206
```

This time, there are $2,000 - 205 - 1 = 1,794$ messages pending in the buffer. As expected, the subscriber can't keep pace with the publisher, and you could potentially cause the JVM to become unstable with out-of-memory errors or degrade in performance.

TIP Runtime statistics about the Camel Reactive Streams are available from JMX. Under the services folder, you can find the `DefaultCamelReactiveStreamsService` MBean, which has JMX operations that return the statistics in tabular format.

ADDING BACK PRESSURE

To avoid such problems, you can use back pressure to prevent dequeuing too many messages from the JMS queue and instead try to keep a pace that's more aligned with the subscriber.

The strategy for back pressure that you can use in this example is to use Camel's `ThrottlingInflightRoutePolicy` in the Camel route, as shown in the following listing.

Listing 20.8 Reactive flow with back pressure using Camel route policy

```
public void configure() throws Exception {  
    ThrottlingInflightRoutePolicy inflight = ①
```

①

Creates route policy

```
new ThrottlingInflightRoutePolicy(); ①  
inflight.setMaxInflightExchanges(20); ②
```

②

Configures policy

```
inflight.setResumePercentOfMax(25); ②  
from("seda:inbox").routePolicy(inflight) ③
```

③

Uses policy in Camel route

```
.delay(100)  
.log("Camel routing to Reactive Streams: ${body}")  
.to("reactive-streams:inbox");  
}
```

To make the reactive flow and the Camel route in this listing flow with a similar pace, you can use Camel's route policy to suspend/resume the route to keep a maximum number of inflight messages. Therefore, you create the

ThrottlingInflightRoutePolicy **①**, which is configured to limit at most 20 inflight messages **②** and resume again at 25 percent of the maximum (= 5 messages). In other words, the rate will be between 5 and 20 inflight messages. To use the policy, you must remember to configure it on the route **③**.

TIP You can find more information about the Camel route policy in chapter 15, section 15.2.3.

You can run this example by executing the following Maven command:

```
cd chapter20/rx-java2  
mvn test -Dtest=CamelInflightBackPressureTest
```

The following output is captured at a similar moment, when we ran the example without back pressure:

```
INFO route1          - Camel routing to Reactive Streams:  
Hello 198  
INFO route1          - Camel routing to Reactive Streams:  
Hello 199  
INFO BackPressureTest - Processing message Hello 184  
INFO BackPressureTest - Processing message Hello 185
```

This time, you can see that when the 200 messages have been routed by Camel, the reactive flow is currently processing message 184. Only 15 ($200 - 184 - 1 = 15$) pending messages are in the reactive buffer. This is because of the back pressure in use.

The following two outputs represent when the back pressure is in use by first suspending the route and a little while later resuming the route:

```
INFO route1          - Camel routing to Reactive Streams:  
Hello 107  
INFO route1          - Throttling consumer: 22 > 20  
inflight exchange by suspending consumer:  
SedaConsumer[seda://inbox]  
INFO BackPressureTest - Processing message Hello 87  
...  
INFO BackPressureTest - Processing message Hello 102  
INFO BackPressureTest - Processing message Hello 103  
INFO route1          - Throttling consumer: 5 <= 5  
inflight exchange by resuming consumer:  
SedaConsumer[seda://inbox]  
INFO BackPressureTest - Processing message Hello 104  
INFO route1          - Camel routing to Reactive Streams:  
Hello 108
```

And when you run the example with 2,000 messages, you can see at the end of the test that the back pressure works as if there are only 19 pending messages in the reactive buffer:

```
INFO route1          - Camel routing to Reactive Streams:  
Hello 1998  
INFO route1          - Camel routing to Reactive Streams:  
Hello 1999  
INFO BackPressureTest - Processing message Hello 1980  
INFO BackPressureTest - Processing message Hello 1981
```

About using back pressure with the throttling route policy

Using `ThrottlingInflightRoutePolicy` as back pressure works by suspending and resuming the Camel route. This works the best when the route is consuming messages from a messaging system such as JMS, AMQP, or Kafka. But if the route is an HTTP service, then the route suspending will cause the HTTP service to be unavailable and return HTTP Status 503 to clients. That may not be desirable, and you should try to scale out your applications in a cluster, as covered in chapters 17 and 18.

If a certain amount of data loss is acceptable, you can configure Camel to use a different strategy than buffering every message.

USING ALTERNATIVE BACK-PRESSURE STRATEGIES

Table 20.1 lists the back-pressure strategies supported by Camel reactive-streams component.

Table 20.1 Back-pressure strategies supported by Camel

Strategy	Description
BUFFER	Buffers all messages in an unbounded buffer. This is the default strategy.
LATEST	Keeps only the latest message and drops all the others.
OLDEST	Keeps only the oldest message and drops all the others.

So far, all the examples have been using the default `BUFFER` back-pressure strategy, which is the first item in table **20.1**. This strategy buffers the messages, which ensures that no data loss occurs. But you learned about the potential danger if the publisher produces messages faster than the downstream

subscribers can process; this can cause the JVM to consume more and more memory, degrade in performance, or eventually run out of memory. Camel supports alternative strategies if message loss can be accepted, which is done by discarding messages from the buffer. When using the LATEST or OLDEST strategy, the discarded messages will be removed from the buffer, and a `ReactiveStreamsDiscardedException` exception is thrown for each message. By throwing the exception, you can use Camel to react and use its error handling to route the message to a dead letter channel, or to log the message, or silently ignore it.

Let's see an example in which you want to process only the latest message and silently ignore the discarded messages. You can run this example from the chapter20/rx-java2 directory as follows:

```
mvn test -Dtest=CamelLatestBackPressureTest
```

The interesting output from running the example is highlighted here:

```
INFO route1           - Camel routing to Reactive
Streams: Hello 193
INFO route1           - Camel routing to Reactive
Streams: Hello 194
INFO LatestBackPressureTest - Processing message Hello 194
INFO route1           - Camel routing to Reactive
Streams: Hello 195
INFO route1           - Camel routing to Reactive
Streams: Hello 196
INFO route1           - Camel routing to Reactive
Streams: Hello 197
INFO route1           - Camel routing to Reactive
Streams: Hello 198
INFO route1           - Camel routing to Reactive
Streams: Hello 199
INFO LatestBackPressureTest - Processing message Hello 199
```

As you can see, the reactive flow processes the latest message that was sent to the buffer. All the other messages are discarded. To avoid causing every discarded message to fail and have its stacktrace logged, you can use Camel's error handler to handle the `ReactiveStreamsDiscardedException` exception by adding the

following line to the Camel route:

```
onException(ReactiveStreamsDiscardedException.class).handle  
d(true);
```

You can also use back pressure from the other side, the consumer side.

20.1.6 CONTROLLING BACK PRESSURE FROM THE CONSUMER SIDE

When Camel consumes messages from a reactive publisher, it has back pressure enabled out of the box. The Camel consumer allows by default at most 128 inflight messages, which are used to determine the number of messages to request from the publisher when requesting for more data. In other words, Camel will at most request up to 128 messages from the publisher.

You can configure the maximum number of inflight messages as an option on the endpoint:

```
from("reactive-streams:inbox?maxInflightExchanges=5")
```

The following listing shows the source code of an example using back pressure on the consumer side.

Listing 20.9 Camel reactive consumer with back pressure

```
public void testConsumerBackPressure() throws Exception {  
    CamelReactiveStreamsService rsCamel =  
    CamelReactiveStreams.get(context);  
  
    String[] inbox = new String[100];  
    for (int i = 0; i < 100; i++) {  
        inbox[i] = "Hello " + i;  
    }  
  
    Flowable.fromArray(inbox) ①
```

①

Reactive flow

```
.doOnRequest(n -> { ②
```

②

Logs every request for more data

```
    log.info("Requesting {} messages", n); ②
})
.subscribe(rsCamel.streamSubscriber("inbox",
String.class)); ③
```

③

Camel reactive subscriber

```
    Thread.sleep(10 * 1000L);
}

protected RoutesBuilder createRouteBuilder() throws
Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("reactive-streams:inbox?
maxInflightExchanges=5 ④
```

④

Camel reactive route

```
&concurrentConsumers=5") ④
    .delay(constant(10))
    .log("Processing message ${body}");
}
};
```

The example uses a reactive flow that publishes 100 messages ①. To be able to see when Camel requests more data, you use doOnRequest to log the number of messages that were requested ②. The flow then uses Camel as the subscriber ③ on the channel named inbox. You can see how to create a Camel route that consumes from this channel ④. The consumer has been

configured with a maximum of five inflight messages. To speed up processing, you've turned on concurrent consumers with a value of 5 also. This makes Camel create a thread pool of five threads, each of which acts as a reactive subscriber and will independently process the messages from the reactive channel. But they'll collectively act together under the limitations of the maximum inflight messages. Because those values are the same, each consumer won't exceed the maximum number of inflight messages.

The example, provided with the source code in the chapter20/rx-java2 directory, runs using the following command:

```
mvn test -Dtest=CamelConsumerBackPressureTest
```

When running the example, you'll notice the following from the output:

```
INFO CamelConsumerBackPressureTest - Requesting 5
messages
INFO route1 - Processing message
Hello 0
INFO route1 - Processing message
Hello 4
INFO route1 - Processing message
Hello 3
INFO route1 - Processing message
Hello 2
INFO route1 - Processing message
Hello 1
INFO CamelConsumerBackPressureTest - Requesting 3
messages
INFO CamelConsumerBackPressureTest - Requesting 1
messages
INFO CamelConsumerBackPressureTest - Requesting 1
messages
INFO route1 - Processing message
Hello 7
INFO CamelConsumerBackPressureTest - Requesting 1
messages
INFO route1 - Processing message
Hello 5
INFO CamelConsumerBackPressureTest - Requesting 1
```

messages

As you can see, the output shows that Camel requests the maximum number of inflight messages at first, and then as it runs the request, it drops down to one or three messages. Often only one message is requested. That's because each consumer thread will refill the buffer when it has processed the message. And because the thread has just completed its own message, the buffer often has room for only one more message.

Doing frequent `request(1)` calls is generally a bad pattern, as this increases the communication costs (chatty network). Therefore, Apache Camel from version 2.20 onward provides the `exchangesRefillLowWatermark` option, which is used as a threshold to trigger when to request more data. The watermark is a percentage of the maximum inflight exchanges and has the default value of 0.25. Running the same example as before with Camel 2.20 onward will demonstrate that Camel doesn't perform any `request(1)` calls anymore.

That's all we could squeeze into this chapter about Camel, Reactive Streams, and the Camel reactive-streams component. This fairly new addition to the Apache Camel project is expected to be taken up by more Camel users when reactive applications start to become more in use. We'll come back to some more thoughts on this in the chapter summary.

At this point, let's move on to the second half of this chapter, which is devoted to Vert.x.

20.2 Using Vert.x with Camel

On the Eclipse Vert.x website, the project describes itself as *a toolkit for building reactive applications on the JVM*. This description has three important points.

First, as a toolkit, Vert.x isn't an application server. Vert.x is just a JAR file (`vertx-core`), so a Vert.x application is an application that uses this JAR file.

Second, Vert.x is reactive and adheres to the Reactive Manifesto (<http://reactivemanifesto.org>), which is highlighted in the following four bullets:

- *Responsive*—A reactive system needs to handle requests in a reasonable amount of time.
- *Resilient*—A reactive system must stay responsive in the face of failures, so it must be designed with failure in mind.
- *Elastic*—A reactive system must stay responsive under various loads. Therefore, it must be scalable.
- *Message driven*—Reactive systems rely on asynchronous messages passing between its components. This establishes a boundary between components that ensures loose coupling, isolation, and location transparency.

Finally, Vert.x applications run on the JVM, which means Vert.x applications can be developed using any of the JVM languages such as Java, Groovy, Scala, Kotlin, Ceylon, and JavaScript. The polyglot nature of Vert.x allows you to use the most appropriate language for the task.

TIP If you’re new to Vert.x, we recommend the free book *Building Reactive Microservices in Java* by Clement Escoffier, which you can download from <https://developers.redhat.com/promotions/building-reactive-microservices-in-java>. This book covers Apache Camel. But we want to give room for Vert.x in this book, as it’s a great toolkit that becomes even greater when used together with Camel.

20.2.1 BUILDING A FOOTBALL SIMULATOR USING VERT.X

What follows next is a true event that happened in the life of Claus:

I hosted a Christmas party with seven of my old friends in

December 2016. When we were younger, from the late 90s to the mid 00s, we'd have these football weekends to watch English football. During a football weekend, the state lottery company would issue a football pool coupon with 13 games. One of these games was the TV game, and out of the remaining 12 games, each of us would select a game. We would then watch the TV game as the other games were played at the same time.

Whenever a goal was scored in any of the games, it would be announced on TV with a *bell* sound. Our rules were simple: if a goal was scored in the TV game, everyone would drink. If a goal was scored in your game, you would drink (bottoms up). To keep the spirit of the old days, a Manchester derby game from February 2004 was selected as the TV game. [Figure 20.3](#) shows a screenshot of the game in action with the goal simulator on the right-hand side.



Figure 20.3 Football simulator playing the TV game with live goal scorer updates on the right

We had a great weekend playing this old game again. Manchester United won 4 to 2 with goals from Scholes and Ronaldo, and two from van Nistelrooy. That was the fun part; now let's talk about how to implement this using Vert.x.

THE ARCHITECTURE

Figure 20.4 illustrates the key components in the architecture.

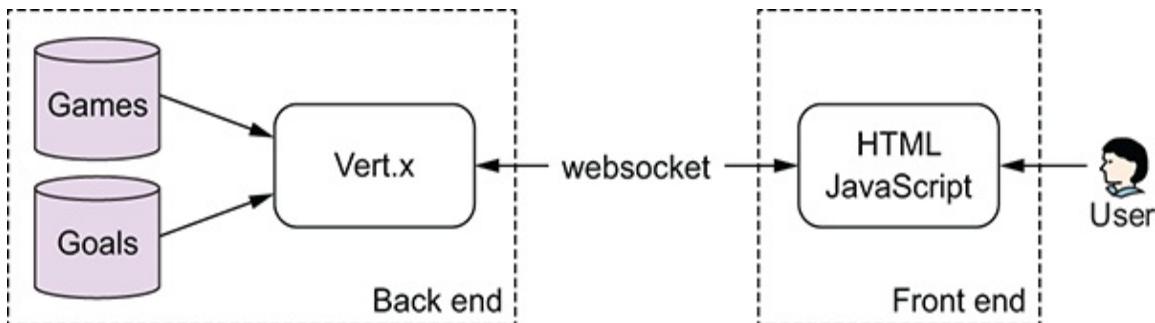


Figure 20.4 In the back end, the Vert.x application loads the games and goals from disk. The front end is an HTML web application with embedded JavaScript that uses WebSocket to communicate with the Vert.x back end.

The back end is implemented as a Vert.x application using Java. Information about the games and the goal scorers is stored in CSV files, which are loaded into the Vert.x application upon startup. The front end is a HTML file with embedded JavaScript. The JavaScript uses the Vert.x JavaScript client that handles all communication to the back end via WebSocket. The entire application is packed together as a single fat JAR, which can run as a Java application on the JVM.

The book's source code contains this example in the chapter20/vertx directory. You can try this by running the following Maven goal:

```
mvn compile vertx:run
```

Then open a web browser to <http://localhost:8080>.

The application is built using Java and HTML code with fascinating parts.

THE JAVA CODE

You develop your Vert.x application in Java code in a *verticle*, which is deployed and running in the Vert.x instance. The verticle is an abstract class that has start and stop methods and easy access to the Vert.x instance itself.

The following listing shows the verticle for the football

simulator.

Listing 20.10 Vert.x verticle for the football simulator

```
public class LiveScoreVerticle extends AbstractVerticle  
{
```

1

The verticle is extending AbstractVerticle class

```
    public void start() throws Exception {  
        Router router = Router.router(vertx);
```

2

Create Vert.x router to set up WebSocket and HTTP server

```
        BridgeOptions options = new BridgeOptions()
```

3

Allowed inbound and outbound traffic on event bus

```
            .addInboundPermitted(new  
PermittedOptions().setAddress("control"))  
            .addOutboundPermitted(new  
PermittedOptions().setAddress("clock"))  
            .addOutboundPermitted(new  
PermittedOptions().setAddress("games"))  
            .addOutboundPermitted(new  
PermittedOptions().setAddress("goals"));
```

```
    router.route("/eventbus/*")
```

4

Route WebSocket to Vert.x event bus

```
        .handler(SockJSHandler.create(vertx).bridge(options,  
event -> {  
    if (event.type() == BridgeEventType.SOCKET_CREATED)
```

```
{  
    vertx.setTimer(500, h -> initGames()); 5
```

5

A new WebSocket client is connected, so initialize list of games to client

```
}  
event.complete(true);  
});
```

```
router.route().handler(StaticHandler.create()); 6
```

6

Add static HTML resources to Vert.x router

```
vertx.createHttpServer()  
.requestHandler(router::accept).listen(8080); 7
```

7

Vert.x router listen on port 8080

```
initControls(); 8
```

8

Initialize game controls and start live score streams

```
streamLiveScore(); 8  
}
```

This football simulator code shows how to build a Vert.x application as a verticle. The `LiveScoreVerticle` class extends `io.vertx.core.AbstractVerticle` **1**. In the `start` method, you have the necessary code to start up, such as creating a Vert.x router for HTTP and WebSocket **2**. Then you set up allowed inbound and outbound communication **3** to the router with the following four event-bus addresses: control, clock, games, and goals. The communication between the back end and front end

uses WebSocket, which you add to the router ④. Whenever a new client is connected, the SOCKET_CREATED event is emitted to the back end, which triggers initialization of the game list ⑤. The HTTP router is also used to service static content such as HTML files ⑥ and is started by listening to port 8080 ⑦. And finally, the game controls and the goal score stream are started ⑧.

The `LiveScoreVerticle` class has more code to initialize the game list and react to game control buttons pushed, game clock advances, and the actual stream of goals. For example, the list of games is initialized as shown in the following listing.

Listing 20.11 Initializing the list of games

```
private void initGames() {  
    try {  
        InputStream is =  
LiveScoreVerticle.class.getClassLoader()  
            .getResourceAsStream("games.csv");  
        String text = IOHelper.loadText(is);      ①
```

①

Loads list of games from classpath

```
Stream<String> games = Arrays.stream(text.split("\n"));  
games.forEach(game -> vertx.eventBus().publish("games",  
game));      ②
```

②

Publishes each game to the event bus

```
} catch (Exception e) {  
    System.out.println("Error reading games.csv file");  
}
```

```
if (clockRunning.get()) {      ③
```

③

Publishes game clock time

```
    vertx.eventBus().publish("clock", "" +  
gameTime.get()); ③  
} else { ③  
    vertx.eventBus().publish("clock", "Stopped"); ③  
}  
} ③
```

The list of games is stored in a CSV file that's loaded ①. Each line in the CSV file is a game that gets published to the Vert.x event bus at the game's address ②. In addition, the game clock state is published ③.

As you can see, it's easy to send messages to the Vert.x event bus using the one-liner code with the `publish` method ②. This is similar to Camel's `ProducerTemplate`, which also makes it easy in one line of code to send a message to any Camel endpoint. But what if you want to consume a message instead? How can you do that from Vert.x?

The football simulator has buttons in the front end that control the game clock, so the user can start and stop the clock. Each time the user clicks those buttons, a message is sent from the front end to the back end using WebSocket on the Vert.x event bus. The following code is all it takes in the back end to set up the consumer:

```
private void initControls() {  
    vertx.eventBus().<String>consumer("control", h -> { ①
```

1

Sets up consumer on the Vert.x event bus control address

```
    String action = h.body(); ②
```

2

The message body contains action to perform

```
        if ("start".equals(action)) {  
            clockRunning.set(true); ③
```

3

Either starts or stops the game clock

```
    vertx.eventBus().publish("clock", "" +  
gameTime.get()); 4
```

4

Publishes the new game clock state back to the front end

```
} else if ("stop".equals(action)) {  
    clockRunning.set(false); 3
```

3

Either starts or stops the game clock

```
    vertx.eventBus().publish("clock", "Stopped"); 4
```

4

Publishes the new game clock state back to the front end

```
    }  
});  
}
```

From the Vert.x event bus, you set up a consumer to listen on the address control **1**, and each message received triggers the handler (using Java 8 lambda style). A Vert.x message also consists of a message body and headers, just like a Camel message. The message body contains what button the user clicked **2**, and you react accordingly to either start or stop the game clock **3**. You then publish the new state back so the web page can react and update its display **4**.

This was the key point from the Java code; let's switch over to the wild west of front-end programming with web frameworks and JavaScript. It's fairly simple, certainly with the Vert.x JavaScript client.

THE HTML CODE

Vert.x allows you to embed web resources such as HTML and JavaScript files in the `src/main/resources/webroot` folder. You have the following files in this folder:

```
└── bell.m4r  
└── index.html  
└── vertx-eventbus.js
```

The `bell.m4r` file is the bell audio that's played when a goal is scored. The `index.html` file is the HTML file that we'll dive into in a moment. And the `vertx-eventbus.js` file is the Vert.x JavaScript client.

The `index.html` file is a plain HTML file with embedded CSS styles and JavaScript. In the `<head>` section, you set up Vert.x as follows:

```
<head>  
  <title>Premier League 2004 Week 7 Livescores</title>  
  <script src="https://code.jquery.com/jquery-  
1.11.2.min.js"></script>  
  <script  
src="//cdn.jsdelivr.net/sockjs/0.3.4/sockjs.min.js">  
</script>  
  <script src="vertx-eventbus.js"></script>  
</head>
```

In this example, we're using the popular JQuery JavaScript library. For WebSocket communication, Vert.x uses SockJS, which provides fallbacks to a simulated WebSocket communication if the web browser doesn't support native WebSocket. The last script is to include Vert.x itself.

In the `<script>` section, you set up the front end to connect to the Vert.x event bus, as shown in the following listing.

Listing 20.12 Using Vert.x in the front end to handle events from the event bus

```
<script>  
  var eb = new EventBus("/eventbus"); 1
```

1

Connects to Vert.x event bus

```
eb.onopen = function () {  
    eb.registerHandler("clock", function (err, msg) { 2
```

2

Reacts when the game clock changes

```
        document.getElementById("clock").innerHTML =  
msg.body;  
});  
  
eb.registerHandler("games", function (err, msg) { 3
```

3

Reacts when the game list is updated

```
var arr = msg.body.split(',');  
var game = arr[0];  
var home = arr[1];  
var away = arr[2];  
  
// more code here not shown  
});  
  
eb.registerHandler("goals", function (err, msg) { 4
```

4

Reacts when a goal is scored

```
if (msg.body === 'empty') {  
    clearScorer();  
    return;  
}  
  
playsound();  
  
// more code here not shown  
})  
};
```

To use the Vert.x event bus from JavaScript, you need to create a new event bus object with the URL to the back end ❶. Then the `onopen` method allows you to register handlers that react when messages are sent to event-bus addresses. In this example, the front end uses three addresses: when the game clock is updated ❷, when the list of games is initialized ❸, and when a goal is scored ❹.

For example, when the game clock is updated ❷, the front end updates the HTML page by setting `innerHTML` to the message body. The game clock is an HTML `<div>` element, as shown here:

```
<div id="clock" class="clock"></div>
```

The source code in [listing 20.12](#) has been abbreviated to not show the JavaScript code that manipulates the HTML elements to update the website.

The source code is located in the `chapter20/vertx` directory. You can try running the example using the following Maven goal:

```
mvn vertx:run
```

Then from a web browser, open `http://localhost:8080`.

The example has been accelerated, so you don't have to wait the full 90 minutes for the football game to end.

If you've been sitting back and relaxing for a while as the goals are pouring in, you may have been enlightened and noticed that the goal simulator isn't using Camel at all. Vert.x is surely an awesome toolkit for building small reactive microservices. But this book is titled *Camel in Action*, so let's update the simulator to use Camel with Vert.x.

20.2.2 USING CAMEL TOGETHER WITH VERT.X

Vert.x and Camel are both small, lightweight toolkits that work well together. [Figure 20.5](#) illustrates this principle, with Vert.x and Camel working together in the same Vert.x application.

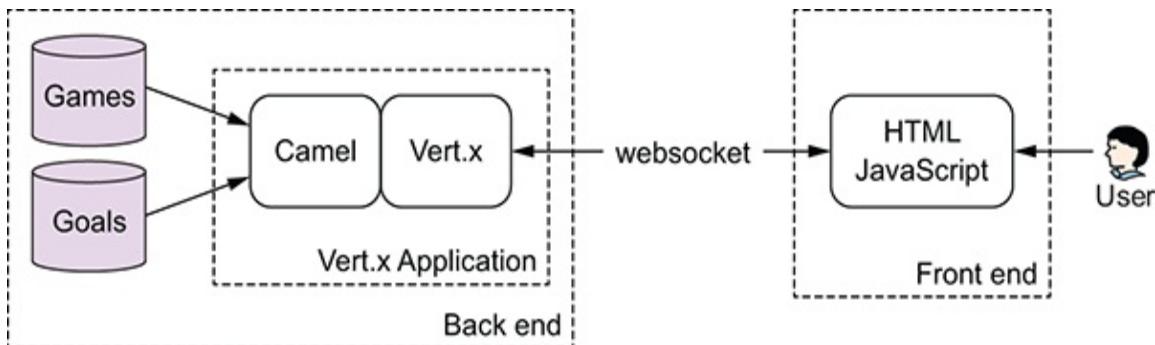


Figure 20.5 Camel and Vert.x working together in the same Vert.x application in the back end. The other parts of the architecture are unchanged.

To use Camel with Vert.x, you need to add camel-core and camel-vertx dependencies to the Maven pom.xml file. In the Vert.x application, you add Camel to the verticle class, as shown in the following listing.

Listing 20.13 Adding Camel to a Vert.x application

```

public class LiveScoreVerticle extends AbstractVerticle {

    private CamelContext camelContext;
    private FluentProducerTemplate template;

    public void start() throws Exception {
        camelContext = new DefaultCamelContext(); 1
    }

```

1

Creates CamelContext

```

        camelContext.addRoutes(new
LiveScoreRouteBuilder(vertx)); 2

```

2

Adds Camel routes that use the Vert.x instance

```

        template =
camelContext.createFluentProducerTemplate(); 3

```

3

Creates ProducerTemplate

```
camelContext.start(); 4
```

4

Starts Camel

```
Router router = Router.router(vertx);
```

5

Sets up Vert.x Router

```
}
```

```
public void stop() throws Exception {  
    template.stop(); 6
```

6

Stops Camel

```
    camelContext.stop(); 6
```

```
}
```

The code to add Camel should be familiar to you. At first, CamelContext is created **1**, and then routes are added **2**. Then you create ProducerTemplate, **3** which will be used by Vert.x to trigger a Camel route. Camel is then started **4**, and the subsequent code sets up Vert.x router **5**. When the Vert.x application stops, you must remember to stop Camel as well **6**.

CALLING CAMEL FROM VERT.X

When a new client connects to the back end, Vert.x will trigger the SOCKET_CREATED event, which you use to obtain the list of games and send to the front end so the website can be updated accordingly. In the previous example, you used Java code to load the game list from a CSV file and send the information using

Vert.x. This time, you're using Camel, so the `SOCKET_CREATED` event uses `ProducerTemplate` to trigger a Camel route by sending an empty message to the `direct:init-game` endpoint ❶, as shown here:

```
router.route("/eventbus/*")
    .handler(SockJSHandler.create(vertx).bridge(options,
event -> {
    if (event.type() == BridgeEventType.SOCKET_CREATED) {
        vertx.setTimer(100, h -> template.to("direct:init-
games").send()); ❶
```

❶

Calling Camel route using the `ProducerTemplate`

```
}
```

```
event.complete(true);
});
```

As you can see, calling Camel from Vert.x is simple; you use regular Camel APIs such as a `ProducerTemplate`. But what about the other way around? How do you make Camel call Vert.x?

CALLING VERT.X FROM CAMEL

Camel provides the `camel-vertx` component that's used to route messages to/from the Vert.x event bus and Camel. The following listing shows how this is done.

[Listing 20.14](#) Using Camel routes to stream live goal scores to Vert.x event bus

```
public class LiveScoreRouteBuilder extends RouteBuilder {

    private final Vertx vertx;

    public LiveScoreRouteBuilder(Vertx vertx) {
        this.vertx = vertx; ❶
```

❶

Injects Vert.x instance in constructor

```
}
```

```
public void configure() throws Exception {  
    getContext()  
        .getComponent("vertx", VertxComponent.class)  
        .setVertx(vertx); 2
```

2

Sets up Vert.x instance on Camel vertx component

```
from("direct:init-games").routeId("init-games") 3
```

3

Routes to initialize game list

```
.log("Init games event")  
.to("goal:games.csv") 4
```

4

Gets list of games from goal component

```
.split(body())  
.to("vertx:games"); 5
```

5

Splits each game and sends to the vertx event bus

```
from("goal:goals.csv").routeId("livescore").autoStartup(false) 6
```

6

Routes to stream live goal scores

```
.log("Goal event: ${header.action} -> ${body}")  
.choice()  
    .when(header("action").isEqualTo("clock"))
```

```
.to("vertx:clock")
```

7

7

Updates game clock

```
.when(header("action").isEqualTo("goal"))
    .to("vertx:goals");
```

8

8

Updates goal score

```
from("vertx:control").routeId("control")
```

9

9

Routes for control buttons

```
.log("Control event: ${body}")
    .toD("controlbus:route?
routeId=livescore&async=true&action=${body}");
}
```

The `LiveScoreRouteBuilder` class is injected with the Vert.x instance ① in the constructor because you need to configure the Camel vertx component to use this instance ②.

Then three Camel routes follow. The first route is used to initialize the list of games ③. The route calls the goal component ④, which is responsible for loading the game list from the filesystem. The front end expects one message per game, hence you need to split the game list before sending to the Vert.x event bus on the game's address ⑤.

NOTE To hide the complexity of loading the game list and streaming live goal scores, we've built a Camel component named `goal`.

The second route is responsible for streaming game-clock and goal-score updates ⑥ that are routed using a Content-Based Router EIP to either the clock ⑦ or goal ⑧ address on the Vert.x event bus. Pay attention to the fact that the route has been configured to not automatically start. You want the user to click the Start button at the front end, which is controlled by the last route ⑨. The control-bus component is capable of starting and stopping routes. Notice that you refer to the `livescore` route using the `routeId` parameter on the `controlbus` endpoint.

TIP Chapter 16 covers much more about the control-bus component in its discussion about managing and monitoring Camel.

The `action` parameter tells Camel what to do, such as start, stop, suspend, or resume the route. This action is triggered from the front end using the following JavaScript functions:

```
startClock = function () {
    eb.send("control", "start");
};
stopClock = function () {
    eb.send("control", "suspend");
};
```

The rest of the code for this example is the goal component, which hides the logic to read the CSV files and stream the game clock and goal updates.

We encourage you to take a moment to look at this example and pay attention to how Vert.x and Camel are loosely coupled and clearly separated.

The only touchpoint between them is the exchange of messages using the Vert.x event bus using the `camel-vertx` component. [Figure 20.6](#) illustrates this principle.

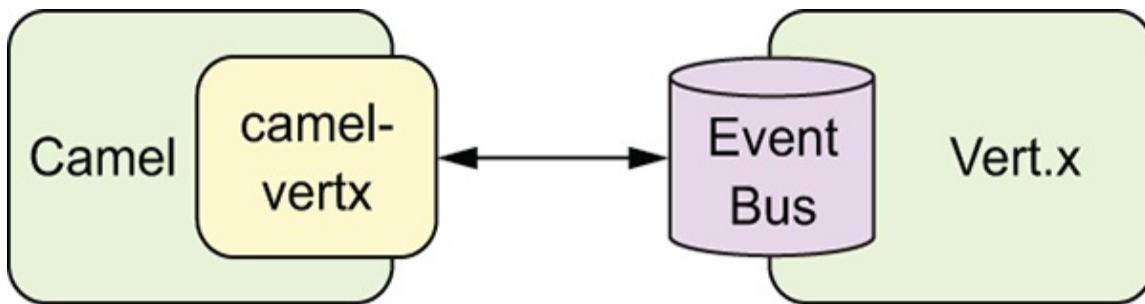


Figure 20.6 Camel and Vert.x exchange messages using the Vert.x event bus. Camel facilitates this using the camel-vertx component.

TRYING THE EXAMPLE

You can find the source code in the chapter20/vertx-camel directory, and you can try the example using the following Maven goal:

```
mvn compile vertx:run
```

Then, from a web browser, open <http://localhost:8080>.

Okay, let's end the goal scoring simulator and conclude our Vert.x coverage in this book with some final words.

20.2.3 SUMMARY OF USING CAMEL WITH VERT.X FOR MICROSERVICES

Building the goal scoring simulator has been a fun ride, using Vert.x and then later adding Camel to the mix. This kind of application runs well with Vert.x, which has great support for HTML and JavaScript clients. Notice how easy it is to exchange data between the Java back end and the HTML front end using the Vert.x event bus. The web front end is a modern HTML5 single-page application that reacts to a live stream of events. This is where Vert.x shines. How does this compare to Camel? Vert.x is focused on reactive applications, and Camel is focused on messaging and integration. It's the combination of the two that gives synergy ($2 + 2 = 5$).

There's a lot of good to say about Vert.x, and we recommend you look at the project and keep an eye on it for the future. Vert.x is reactive, asynchronous, and nonblocking; it puts the burden

on the developer to understand its APIs. This takes time to master and grasp. Camel, on the other hand, hides a lot of that complexity. As a developer, you can get far with Camel routes and configuring Camel components and endpoints. Therefore, we recommend you study the Vert.x APIs and programming model if you take up using Vert.x. A good place to start is with the book *Building Reactive Microservices in Java* by Clement Escoffier, which you can free download for free from <https://developers.redhat.com/promotions/building-reactive-microservices-in-java>.

20.3 Summary and best practices

We're ending this bonus chapter on a high note with coverage of a complex but interesting topic of reactive applications and systems. Although reactive principles and frameworks have been around for many years, they've only recently gained momentum and attention from developers (we're not talking about the ninja developers who jump from hipster technology to hipster technology).

The addition of lambdas and streaming API in Java 8 has also helped Java developers get exposed to and become more familiar with the streaming style of programming. Another popular framework that would push in this direction is the Spring Framework, which include a reactive API from version 5.

As you've seen in this chapter, Apache Camel is also striding into the reactive world with the Reactive Streams component. The Camel team has designs for the Camel 3.x architecture to offer two routing engines and APIs:

- *Classic routing engine*—The classic routing engine as of today.
- *Reactive routing engine*—A reactive engine based on the Reactive Streams API and pluggable runtime reactive library such as RxJava or Reactor Core.

By having both routing engines side by side, Camel allows users

to pick and choose what best suits their situations. This also allows ample time for the new reactive engine to be developed and matured over the years, with valuable feedback and influence from the community.

We do want to say that reactive streaming APIs and reactive flows can be difficult to learn and understand. If you decide to use this for serious work, make sure to take extra time to learn and experiment.

As usual, here's a bulleted list of the highlights and our thoughts:

- *Reactive applications are complex*—Learning, using, and developing reactive applications is more complex and harder than regular applications. Taming the asynchronous beast isn't easy—especially the RxJava library, which has a lot of greatness but is also harder to get working and understand once you move past the beginner stage. If you become more serious, try to find good online material or a book. We recommend *Reactive Programming with RxJava* by Tomasz Nurkiewicz and Ben Christensen (O'Reilly, 2016).
- *Vert.x has potential*—Keep an eye on the Vert.x project, as it's good and has great potential. It's a small, lightweight tool to build reactive applications. The camel-vertx component enables you to easily use the many Camel components in your Vert.x applications.
- *Reactive, reactive, reactive*—When Docker came out, it was Docker, Docker, Docker. You may hear more and more about reactive systems, streams, and programming. It's not a game changer requiring you to drop everything and rewrite your applications to these new systems, frameworks, and toolkits.

This is the end of the first bonus chapter. The second bonus chapter covers Camel and the Internet of Things (IoT). Neither Claus nor Jonathan is a domain expert on IoT, so we've invited Henryk Konsek to be the guest author for this chapter. This will be the last word you hear from Claus and Jonathan; Henryk, the

floor is yours.

21

Camel and the IoT

BY HENRYK KONSEK

This chapters covers

- Basic introduction to the Internet of Things (IoT)
- Suggestions for purchasing base IoT hardware
- IoT architecture
- Reasons for using Camel for IoT applications
- Camel gateway-to-data-center connectivity
- Integrating Apache Camel and Eclipse Kura

The *Internet of Things* (IoT) is a term used to describe a certain class of distributed IT systems that work with clients located on distributed devices connected to back-end messaging systems. Imagine a centered back-end system with many devices connected to it (like the one in [figure 21.1](#)); this is pretty much what IoT is.

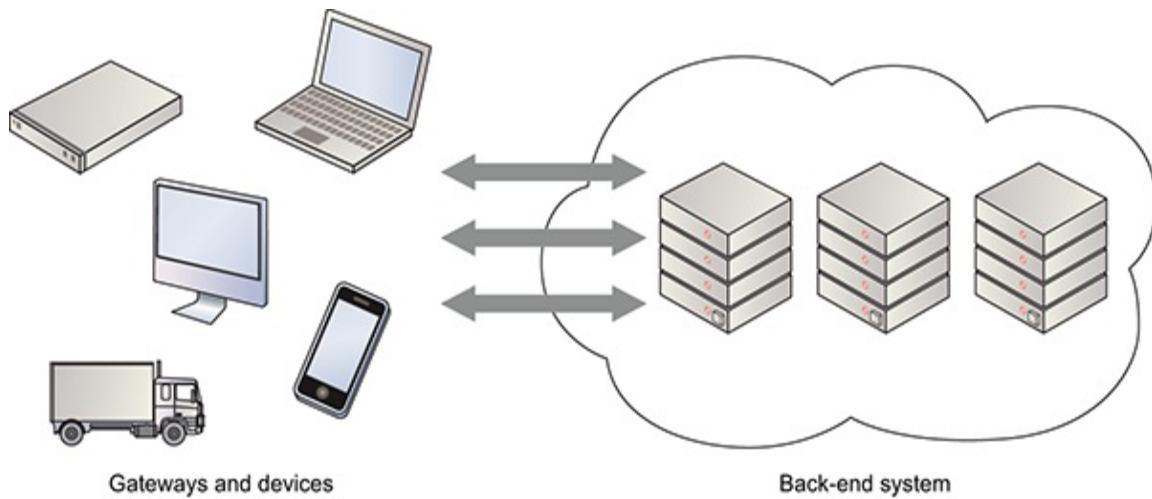


Figure 21.1 General view of the IoT architecture

These connected devices can range from mobile phones, tablets, and Raspberry Pi boards to something as large as a car. The IoT is important for our industry, because the number of connected devices in the world is growing exponentially, and we need to find a way to deal with this kind of scale of distributed clients. In addition, market predictions indicate that sooner or later the majority of developers will be involved in an IoT project of some sort.

Many of us associate the IoT with futuristic devices such as drones and robots, with hipster startups, or with Raspberry Pis. These may be true examples, but the reality is that IoT usually isn't as spectacular as we might expect. The IoT points to a future of boring verticals such as industry, army, intelligence, automotive, home automation, city infrastructure, and medical equipment. And in the majority of cases, end consumers won't even know that they're interacting with an IoT system. For example, all our dishwashers sooner or later are going to send information about their use to their manufacturers. Home furnaces are definitely not as spectacular as drones or robots, but we'll save money on furnace maintenance, as it might alert us whenever our manufacturer detects disturbing patterns in usage.

This chapter provides a gentle introduction to the Internet of Things in general. Then you'll see how Camel can help you build an IoT-class system. Finally, we'll focus on using Apache Camel

and Eclipse Kura to greatly simplify providing production-grade IoT gateway solutions. But before proceeding to the details of IoT architecture, let's go shopping. You need some IoT equipment before digging into the IoT world.

21.1 The Internet of Things shopping list

This section provides an opinionated list of hardware you might want to purchase in order to start playing with Camel and the IoT. Although almost every IoT enthusiast could provide another list and argue that it's better than this one, my list is based on the following important factors:

- *Price*—You don't want to spend an excessive amount of money just to start a journey with the Internet of Things. Experiments ought to be cheap so you can play and innovate without special financial concerns.
- *Functionality*—You'll want hardware with many sensors and other features that allow you to experiment with numerous test scenarios. The more sensors (such as temperature or humidity sensors or gyrometers) and actuators that your hardware has, the more exciting experiments you can do with it.
- *Availability in various countries*—A myriad of hardware is on the market, but not all devices are easily available in every country. I've tried to select those devices that can be easily purchased in and shipped to the most places in the world.
- *Number of tutorials and online resources available*—You want to be sure that hardware you purchase is popular enough that you can easily search online for tutorials and answers to your questions and problems. Some hardware is better than other hardware, but for learning purposes, you should use the most popular ones.

21.1.1 RASPBERRY PI

First of all, you need an IoT gateway. I'll explain this term later

in this chapter; for now, you just need to know that you need a small computer board that can be used to run a small Camel application. I recommend purchasing a Raspberry Pi for this purpose.

Raspberry Pi is a small computer with 1 GB of RAM and a decent CPU unit (ARM based). The board size is slightly bigger than a pack of cigarettes, which is pretty small for a computer. Raspberry Pi is extremely popular in the *maker* community, so you can find a gazillion tutorials related to this awesome piece of hardware.

To be specific, I recommend buying the Raspberry Pi 3 model B (www.raspberrypi.org/products/raspberry-pi-3-model-b/), which has significant advantages over the previous editions of the board. In particular, it has the following:

- 64-bit CPU
- Embedded Wi-Fi and Bluetooth Low Energy (BLE) units

The price of a Raspberry Pi 3 is about \$45. You may also consider buying a case for your Raspberry Pi board (www.raspberrypi.org/products/raspberry-pi-case/), which costs less than \$8 and is a fancy plastic cover for your board. It's not necessary to have a case for your Raspberry Pi, but a case can protect your board from physical damage. Last but not least, the Pi looks pretty spiffy in a case, compared to a naked board.

21.1.2 SD CARD FOR RASPBERRY PI

Raspberry Pi doesn't come with any persistent storage. Instead, it provides a slot for a micro SD card, which can be inserted to serve as Raspberry Pi's disk. An SD card is also used to install an operating system (www.raspberrypi.org/documentation/installation/installing-images/), which the board can bootstrap on startup. Raspberry Pi supports a bunch of Linux distributions, including Raspbian (www.raspberrypi.org/downloads/raspbian/) and Fedora (https://fedoraproject.org/wiki/Raspberry_Pi). I recommend

buying at least an 8 GB SD card, especially if you plan to install Raspbian. Prices of decent 8 GB micro SD cards start at \$5.

21.1.3 POWER BANK FOR RASPBERRY PI

Raspberry Pi is powered using a micro USB cable. In practice, you can use your Android phone charger (or any other micro USB cable connected to a laptop, for example) to power your Raspberry Pi unit. Although it's perfectly fine to run your Pi by using one of those cables, I highly recommend purchasing a power bank and powering your board with it. A *power bank* is a rechargeable battery with USB ports.

If you connect your Raspberry Pi to a power bank, you can take it anywhere you want; you aren't limited by a cable plugged into a power socket. You can even take your Pi outdoors—for example, to measure how temperature changes when you go outside your home.

I don't recommend any particular power bank producer or model. Prices of power banks start at a few bucks, but you should invest in a unit with as much power capacity as possible. The more power capacity your power bank has, the longer Raspberry Pi can run connected to it without an additional charging cycle.

21.1.4 CAMERA FOR RASPBERRY PI

Another optional, but highly recommended, item for IoT developer wannabes is a camera unit. A Raspberry Pi camera, which can be purchased for less than \$25, is a useful piece of equipment. It allows you to collect video streams and analyze or store those streams on your board. This is an excellent unit for creating simple applications, demonstrating how Raspberry Pi can be used to detect motion, process images, recognize objects in a stream of video, and so forth.

21.1.5 TI SENORTAG

You may already have your Raspberry Pi gateway device ordered, but you won't be able to collect any data with it. You need sensors to do that. Although it's perfectly fine to purchase a

bunch of sensors from your favorite electronics store and connect those to your Raspberry Pi via its general-purpose input/output (GPIO) interface, wiring all your sensors to your Pi board requires an electronics background and effort. A great alternative is raw sensors, which can be efficiently used for learning purposes.

Texas Instruments SensorTag is a small device containing a bunch of sensors and wireless connectivity units (www.ti.com/ww/en/wireless_connectivity/sensortag2015/?INTC=SensorTag&HQS=sensortag).

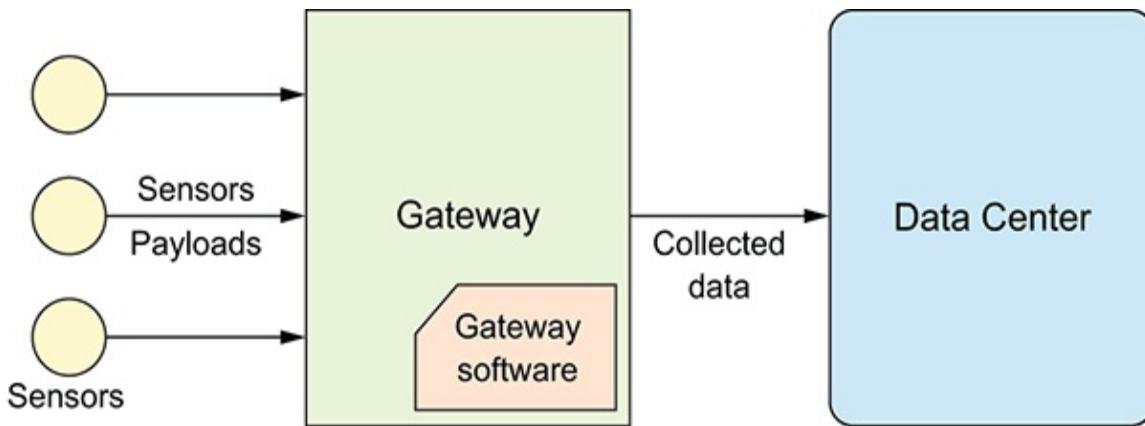
The TI SensorTag unit can be purchased for about \$30. The device, powered by a tiny battery, brings an impressive array of sensors for that price, including a temperature sensor, humidity sensor, and gyroscope, among others. The online SensorTag User Guide provides details (see http://processors.wiki.ti.com/index.php/SensorTag_User_Guide#Sensors). You can choose the kind of wireless connectivity you're interested in when buying a unit, but choosing a model with a Bluetooth Low Energy module is a great match for Raspberry Pi 3, which happens to provide BLE connectivity out of the box as well.

In practice, purchasing a BLE-enabled SensorTag unit allows you to send telemetry readings straight into your Raspberry Pi unit and process those readings with Camel. A great match indeed! After you purchase these useful hardware devices, you're ready to dig into the IoT architecture to see how this hardware can be connected to your Camel routes.

21.2 The Internet of Things architecture

Before you dig into Camel in the context of the Internet of Things, let's look at a generic IoT architecture example so you can understand the kinds of components needed to create an operational system for connected devices. You need this knowledge to understand where Camel can help you and where it can't.

Architectures of an IoT system can be less or more complex, but we can extract a main pattern from the majority of applications of this class. [Figure 21.2](#) presents a typical IoT system architecture.



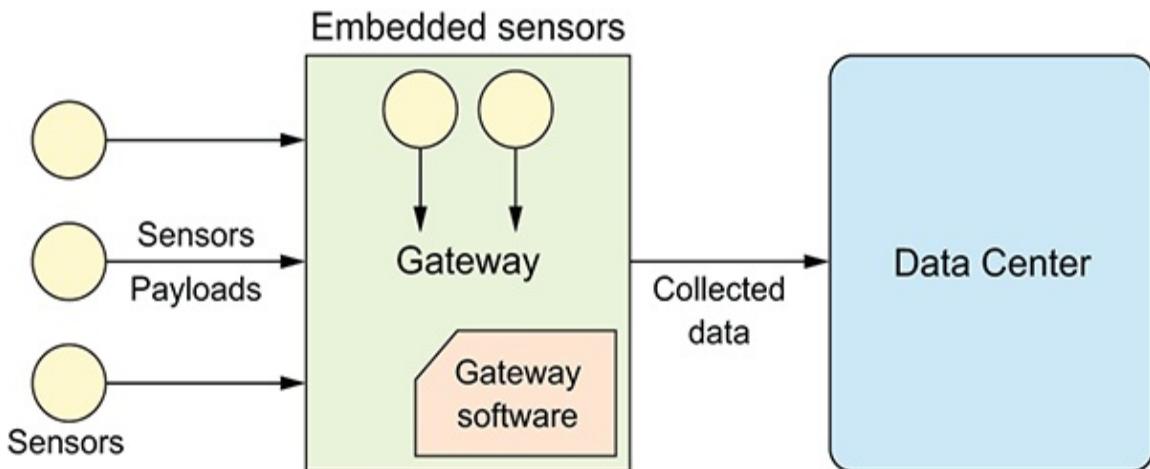
[Figure 21.2](#) Internet of Things application architecture

First of all, the system has sensors (visible on the left of [figure 21.2](#)). *Sensors* are small devices that can collect information from an environment. This information may be readings of temperature, air humidity, vibration level, sound, or video from a digital camera. Sensors usually run embedded software and are too small to run Java and Camel. Moreover, sensor devices are usually too small to handle the whole TCP/UDP stack by themselves (they don't provide TCP/UDP connectivity, or *internet* connectivity). Instead, sensors use more lightweight and limited protocols such as BLE, ZigBee, or Z-Wave. Those protocols provide wireless connectivity limited to an effective range of less than 100 meters.

This is where the middle part of [figure 21.2](#) comes in. You need a piece of hardware that can use short-range wireless protocols such as BLE, collect payloads from sensors, and forward these payloads to a data center using TCP or UDP connectivity via Wi-Fi or Ethernet interfaces. This hardware is called a *gateway*. It's a board similar to Raspberry Pi, but usually with a better hardware specification. The gateway is more powerful and can usually run Java Virtual Machine and take advantage of Camel (more on this later in this chapter).

The last piece of the IoT architecture puzzle is data center connectivity (represented on the right side of [figure 21.2](#)). There's no point in collecting data on a gateway if you don't send the information you've gathered into a back-end service for further analysis and processing. IoT gateways usually rely on TCP/UDP protocols such as MQTT, AMQP, CoAP, LWM2M, and HTTP/REST for data center connectivity. The gateway's task is to enqueue collected data, preprocess it, filter it if needed, and finally send it to a data center back-end endpoint.

Keep in mind that sensors aren't always connected to a gateway using wireless protocols (such as BLE or Z-Wave). It's common to wire sensors directly into gateway hardware with physical interfaces such as GPIO. [Figure 21.3](#) shows how the architecture might look in such a scenario. As you can see, sensors are not only connected to a gateway via wireless protocols, but also embedded in the gateway itself.



[Figure 21.3](#) IoT application architecture with sensors embedded in a gateway

21.3 Why Camel is the right choice for the IoT

Now you have a sense of the general architecture of IoT applications. But you might still be wondering whether Camel is the right choice for these applications and the real reason to use it for IoT purposes. This section covers the most important areas

where Camel functionalities excel in the context of the Internet of Things to bring real value for developers.

Before we dig into the details of using Camel in IoT scenarios, let's take a look at those reasons from 10,000 feet:

- Many Camel components are available out of the box.
- Data formats are supported by Camel.
- Redelivery capabilities.
- Throttling support.
- Possibility of performing content-based routing.
- Support for client-side load balancing.
- Runtime flow control provided by the Camel control-bus component.

Let's discuss those reasons in more detail.

21.3.1 COMPONENTS

Camel comes with more than 200 components that can be used to connect to the myriad of protocol endpoints. You can use Camel components on your gateway devices to connect to a data center using the protocol that's the best choice for your device connectivity scenario: AMQP, MQTT, REST, CoAP, Kafka, or many others. Having out-of-the-box components that can be used to handle endpoint connectivity is a huge advantage of Camel, as it allows you to reduce the amount of boilerplate code you need to create, test, and maintain in order to connect your devices to the outside world.

21.3.2 DATA FORMATS

Connectivity between your gateway device and your data center is one thing, but it's essential to understand the encoding of a message sent by your field device. Camel data formats can be used on the gateway side to encode the message appropriately for your back-end system. On the server side, Camel decoders

can be used to deserialize the message via a proper message format: AVRO, JSON, CSV, or others.

21.3.3 REDELIVERY

As stated in this book, Camel provides support for message redelivery when errors occur in data processing. This feature happens to be one of the most wanted functionalities of IoT systems. Because connected devices usually operate on flaky and highly unreliable network connections, it's common to encounter intermittent issues with endpoint connectivity.

IoT systems should be designed to recover gracefully from interruptions of message flow and should attempt to deliver messages later, when connectivity is back again. Message redelivery is one of the key pieces of the error-handling puzzle, and it's great to have it included in Camel out of the box.

21.3.4 THROTTLING

Ideally, you want your sensors to produce telemetry data at a fixed pace—for example, reading temperature every second, or ideally, being notified whenever temperature changes.

Depending on the sensor type you're reading data from, you might from time to time receive more data payloads than you expected. Because IoT gateway devices aren't as powerful as regular server machines, you may end up with sensor readings overflowing your device. You need a mechanism that protects against such telemetry data peaks. You need a way to tell your gateway to accept at most N messages per second. For example, you may want to send at most one temperature reading per second and drop all other payloads (and definitely don't send the other payloads to the data center).

Creating such reliable throttling logic from scratch may be time consuming. Fortunately, Camel comes with Throttler EIP that can be used to limit the data flow. Creating throttling rules in Camel is as simple as the following snippet, which demonstrates code that you might deploy into your gateway device in order to send temperature readings stored on its local

filesystem:

```
from("file:/var/temperature?delete=true")
    .throttle(1).timePeriodMillis(1000)
    .to("paho:temperature");
```

The preceding example reads telemetry payloads stored on a local filesystem and sends that data into an MQTT broker by using the Eclipse Paho component for Camel. Even if temperature payloads are generated quickly, Camel forwards only one message per second and drops all the others.

21.3.5 CONTENT-BASED ROUTING

Some IoT gateways are simple telemetry proxies. They take messages from a sensor and send them to a data center service. In many cases, you want to add some intelligence into your gateway, so you can make it smarter.

For example, you might be interested in filtering out all temperature readings that don't exceed a certain threshold. Imagine, for example, that you want to notify a data center alarm service only when a gateway detects that a temperature in a factory where a gateway is installed is higher than 30 degrees Celsius.

Another scenario that requires the gateway to be smart is a connected car that has to send important telemetry information to a data center using a paid Global System for Mobile Communications (GSM) connection. The same car could send low-priority information, but only when it's within the area of the owner's home, connected to a fast and cheap Wi-Fi network.

This kind of intelligence requires you to analyze the content of each message and define rules to react accordingly. Camel provides excellent support for defining these kinds of rules via content-based routing.

21.3.6 CLIENT-SIDE LOAD BALANCING

As mentioned, IoT gateways are often connected to highly unreliable networks. Another common scenario is to have an IoT

gateway installed in a vehicle that periodically loses connectivity to its network. In such scenarios, it's the gateway's responsibility to attempt to redeliver a message.

As connected vehicles are moving from one destination to another, it's sometimes desirable to attempt to connect to another back-end service when connecting to the first one fails. This kind of behavior can be achieved by using the Camel load balancer EIP. This kind of communication pattern, called a *Circuit Breaker*, can be easily implemented using Camel core features. Also keep in mind that Camel provides a dedicated Hystrix EIP that implements the Circuit Breaker pattern using the popular Hystrix library from Netflix.

21.3.7 CONTROL BUS

Another useful piece of Camel is called the *control bus*. It allows you to dynamically enable and disable certain parts of your routing logic. Is it useful for an IoT? Imagine that your connected vehicle is supposed to flush data cached on its local storage only when it's within the range of a trusted Wi-Fi network. The Camel control bus allows you to enable data synchronization logic only when the device gets close to a Wi-Fi network and to disable it when you're out of range.

21.4 Gateway-to-data-center connectivity

You already know what an IoT architecture could look like. You also have a general understanding of the way Camel can be applied to connected device applications. Now let's focus on the connectivity between a gateway device and data center, as this is an area where Camel can be even more useful.

21.4.1 UNDERSTANDING THE ARCHITECTURE

As already discussed, one of the areas where Camel excels when it comes to the IoT is in communication between a data center and a gateway device. This particular feature of Camel requires extra attention, because the way your messages are transferred

from the field into your remote servers is one of the most important pieces of your IoT architecture. Figure 21.4 demonstrates a simplified architecture for an IoT application. The communication bits are indicated by the circle.

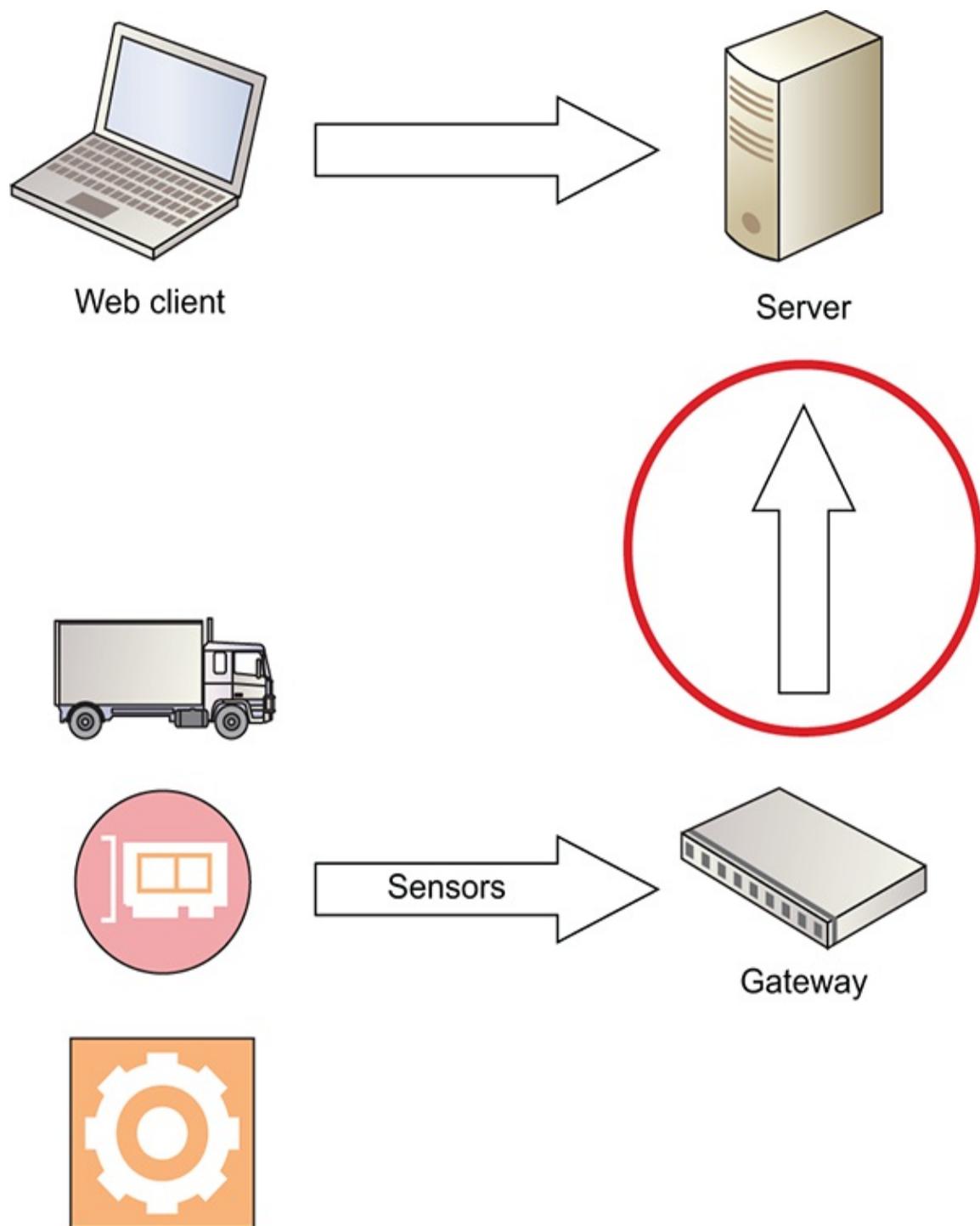


Figure 21.4 Internet of Things connectivity diagram

Protocols working in this layer of an IoT solution are usually TCP-based, but some notable exceptions occur (for example, CoAP and LWM2M are UDP-based). This is good news for backend developers because they can reuse their existing expertise in working with TCP-based messaging protocols and apply it against a gateway device.

21.4.2 CHOOSING A PROTOCOL

In general, you can use any TCP/UDP protocol to transfer data between a gateway and a data center. The choice of protocol should apply the *use the right tool for the job* principle: you shouldn't assume up front that any single protocol is better than another for gateway connectivity. Keeping that in mind, I'll focus on three protocols supported by Camel: AMQP, MQTT, and REST/HTTP, are the most generic and popular gateway connectivity solutions.

AMQP

One popular choice for connectivity between a gateway and data center is Advanced Message Queuing Protocol (AMQP) 1.0. AMQP message overhead is a bit larger than MQ Telemetry Transport (MQTT) messages, but the protocol is much richer in features and better supported on the back end of the system. Many companies provide scalable AMQP message brokers (for example, Azure Service Bus by Microsoft, or A-MQ by Red Hat), whereas large-scale MQTT back-end offerings are limited.

As for message-size overhead, it's the metadata you need to add to every message that matters. For example, HTTP is considered “chatty” because it adds many verbose plain-text headers and instructions for every request. Message overhead is important, especially for gateway devices using paid GSM plans for data transfer; in this scenario, every extra byte added to each message may count. AMQP messages have a reasonable overhead, but it's not significant. I'm talking about AMQP 1.0, not AMQP 0.9 or earlier. This is an important distinction, because AMQP 1.0 is different from its earlier versions.

The main advantages of using AMQP for gateway-to-data-center connectivity are as follows:

- Good scalability of your messaging infrastructure
- Small message overhead
- Request/reply communication support
- Built-in type system that may additionally reduce message overhead
- Flow-control support (dealing with gateways generating too many messages)
- Low-latency peer-to-peer communication support

To use Camel AMQP on your gateway device, you can use the Camel AMQP component. First, add the Camel AMQP JAR to your application:

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

After adding the Camel AMQP JAR into your classpath, register the AMQP component in your Camel context. In particular, you need to specify AMQP broker connection credentials at this stage:

```
import org.apache.camel.component.amqp.AMQPComponent;
...
CamelContext camelContext = ...;
AMQPComponent amqp =
AMQPComponent.amqpComponent.amqpComponent("amqp://localhost
:5672");
camelContext.addComponent("amqp", amqp);
```

Starting from this point, you can send your telemetry data (for example, temperature values read from a sensor and stored on a gateway filesystem) to the remote AMQP endpoint:

```
from("file:/var/temperature?delete=true")
    .marshal().json(JsonLibrary.Jackson)
```

```
.to("amqp:temperature");
```

This demonstrates how to create a Camel route that consumes files from the local filesystem of the gateway device and sends those converted into a JSON payload to the AMQP destination named temperature.

Before executing the preceding snippet, be sure to add the Camel Jackson JAR file into your classpath. The file is needed to serialize the payload into JSON format. The following snippet demonstrates how to do this:

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

You may be wondering how temperature readings end up in the /var/temperature directory. Unfortunately, this kind of code is highly hardware dependent. You can use the GPIO interface to read from temperature sensors—for example, DS18B20 (www.sparkfun.com/products/245) connected directly to your gateway board. You can also use BLE to read temperature values from sensors supporting BLE, such as TI SensorTag. Because this chapter is too short to focus on any particular sensor details, I'll assume your temperature readings are persisted to a kind of storage on your gateway disk. Persisting sensor readings on your local disk before forwarding those to a data center is common practice.

MQTT

The other alternative for gateway-to-data-center connectivity is the MQTT protocol. Let's see how it compares to AMQP.

MQTT stands for *MQ Telemetry Transport* and is one of the most popular TCP protocols for IoT data center connectivity. The main advantage of the MQTT protocol in the context of IoT applications is that metadata overhead per message is small (just a few bytes). As mentioned in the previous AMQP section, the

size of a message is an important factor for connected devices, as the latter often rely on mobile GSM connectivity.

Telecommunication providers charge GSM plan users based on their data consumption, so you want to be sure that your messages don't send unnecessarily large payloads over the wire.

The main advantages of the MQTT protocol are as follows:

- Small message overhead.
- Some devices already support embedded MQTT. These devices can send and receive MQTT messages even though the devices can't run Java applications. You can expect more devices like this to be available in the future.
- Many resources and tutorials related to MQTT are available online.

If you're interested in open source MQTT brokers, you should look at the Eclipse Mosquitto and Apache Artemis projects. Also, Vert.x MQTT is a great brokerless alternative for the MQTT back end.

Camel comes with a component supporting an excellent MQTT client library: Eclipse Paho. To use the Camel Paho component on your gateway device, add the following JAR to your application:

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

Starting from this point, you can send your telemetry data (for example, temperature values read from a sensor and stored on a gateway filesystem) to a remote MQTT endpoint. The only things you need to provide to Paho are network coordinates of your target MQTT broker (network address and port number). The easiest way to start using a Paho component is to configure the broker directly in the endpoint URI:

```
from("file:/var/temperature?delete=true")
```

```
.marshal().json(JsonLibrary.Jackson)
.to("paho:temperature?
brokerUrl=tcp://iot.eclipse.org:1883");
```

An alternative to URI-based configuration is configuring a component instance and registering it directly into the Camel context:

```
import org.apache.camel.component.paho.PahoComponent;
...
CamelContex camelContex = ...
PahoComponent paho = new PahoComponent();
paho.setBrokerUrl("tcp://iot.eclipse.org:1883");
camelContex.addComponent("paho", paho);
```

In this case, you don't have to specify connection coordinates in your route, but you can send messages to the "paho:topicName" endpoint, as shown here:

```
from("file:/var/temperature?delete=true")
.marshal().json(JsonLibrary.Jackson)
.to("paho:temperature");
```

This demonstrates how to create a Camel route that consumes files from the local filesystem of the gateway device and sends those converted into a JSON payload to the MQTT destination named temperature. Before executing the preceding example, be sure to add the Camel Jackson JAR to your classpath.

Please refer to the preceding AMQP section in order to understand how temperature payloads can end up in the /var/temperature directory.

REST/HTTP

You already know how to use the AMQP and MQTT protocols to send messages straight into your messaging broker. That's great, because messaging-oriented solutions are usually the best choice for IoT applications. The primary reason why is that messaging protocols usually are better suited for traffic consisting of many small messages. Also, messaging protocols support not only outbound traffic from a gateway to a data center, but also

inbound traffic (receiving messages from a data center without the need to poll any endpoint). This is particularly important when your gateway needs to receive a command from a data center (for example, to be restarted or to upgrade its software to a more recent version).

Keeping all these points in mind, we can safely say that REST/HTTP usually isn't the best fit for IoT data center connectivity. The message-size overhead of REST/HTTP is rather significant, as HTTP is a chatty, text-based protocol. This is a real disadvantage for GSM-based connectivity. Because of the wide adoption of REST in server-side programming, using HTTP is a popular choice for gateway-to-data-center connectivity. This is especially the case when a gateway device doesn't generate too many network calls to the data center or when a gateway doesn't rely on GSM connectivity (for example, when the gateway is located near the factory floor to help with automation of the industrial process).

Camel comes with several components for HTTP clients; this example will focus on one of them. We'll use a Netty-based HTTP client. To make this client available for your gateway application, add the following line to your Maven pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty4-http</artifactId>
</dependency>
```

From this point, you can send your telemetry data (for example, temperature values read from a sensor and stored on a gateway filesystem) to the remote REST/HTTP endpoint:

```
from("file:/var/temperature?delete=true")
  .marshal().json(JsonLibrary.Jackson)
  .to("netty4-http:http://my.app.com/temperature");
```

This demonstrates how to create a Camel route that consumes files from the local filesystem of the gateway device and sends those converted into a JSON payload to the HTTP URL <http://my.app.com/temperature> using the POST HTTP method.

Before executing the preceding example, be sure to add the Camel Jackson JAR to your classpath.

Please refer to the previous AMQP section in order understand how temperature payloads can end up in the /var/temperature directory.

21.5 Camel and Eclipse Kura

Although deploying a standalone fat JAR application into your gateway device is a perfectly valid solution, it may not be enough for more sophisticated scenarios.

A problem with the fat JAR approach is that it's easy to deploy it into a data center using SCP/SSH, Chef, or Ansible, because server-side resources are supposed to be available all the time. You take your JAR and deploy it into a given server. If you need to upgrade your application, you deploy another version of a JAR. You don't expect your data center server to be out of the network for a certain period of time. Unfortunately, this kind of scenario is common for IoT devices. It's your device's responsibility to attempt to connect to a back-end service and ask for potential software updates. The process of installing updates into a remote device is an over-the-air (OTA) update.

Another issue with IoT software upgrades is that usually you'll want to avoid restarting the whole device during the software upgrade process in order to keep your operations continuous. The fat JAR approach, as convenient as it is, doesn't allow you to restart only certain message flows of your application.

As you can see, updating your gateway software without any additional tool dedicated for this purpose can be challenging. An interesting project that tries to solve such issues is Eclipse Kura. Kura is a low-footprint OSGi server dedicated for gateway devices. Camel provides official support for Eclipse Kura, which provides an opinionated way of deploying Camel routes into a Kura server.

The usual reason to deploy Camel routes into Eclipse Kura is

to provide enterprise integration pattern support for the gateway. The other reason is to provide a myriad of Camel components for Kura. An example of integrating Camel and Kura is installing Kura on the Raspberry Pi board, reading temperature from a sensor installed into that Raspberry Pi using Kura services, and finally, forwarding a current temperature value to your data center service using Camel routes.

Figure 21.5 shows the general architecture of Camel and Kura integration.

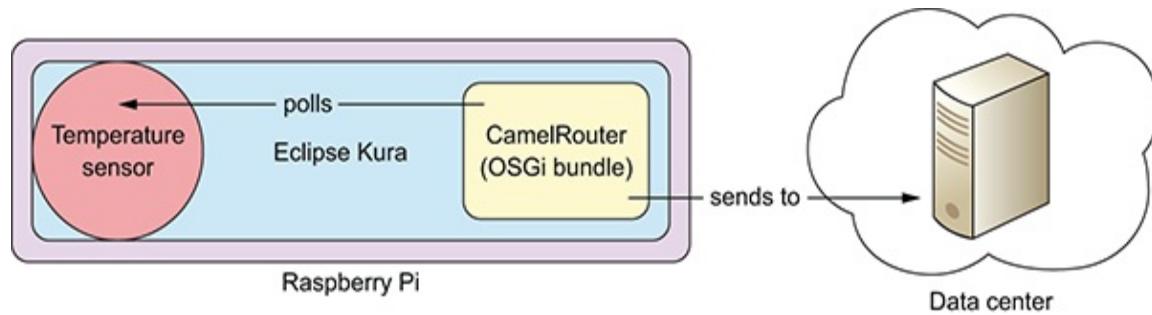


Figure 21.5 Apache Camel and Eclipse Kura integration architecture

Eclipse Kura is installed on the board—for example, on Raspberry Pi, Camel routes are installed as OSGi bundles into the Kura server. Those routes are responsible for using Kura sensor APIs and forwarding messages to a data center.

21.5.1 STARTING KURA IN EMULATOR MODE

It's outside the scope of this book to guide you in installing Kura on a real device board, such as Raspberry Pi. Instead, I'll describe how to run Kura in a board emulator mode as a Docker container. The latter approach is the easiest way to become familiar with Eclipse Kura. If you're interested in installing Kura on a real IoT board, see the official Kura installation guide (<http://eclipse.github.io/kura/intro/raspberry-pi-quick-start.html>). The Kura emulator “pretends” that it's installed on the Raspberry Pi board and allows you to start playing with the Kura web UI and Camel.

Docker is a virtual container management system that allows you to download images containing runnable software and start

those on your local machine. I won't provide detailed installation instructions, and I assume you have Docker installed on your computer.

Let's start your local instance of Kura server. To run the Kura emulator, execute the following Docker command:

```
docker run --name=kura -d -p 8080:8080 ctron/kura-emulator
```

21.5.2 DEFINING CAMEL ROUTES USING THE KURA WEB UI

When you have the Kura emulator running as a background Docker process, open your favorite web browser and navigate to <http://localhost:8080>. You'll be prompted to provide a username and password. Type the username `admin` and password `admin`. After providing valid credentials, you'll see the Kura web UI, which should look similar to [figure 21.6](#).

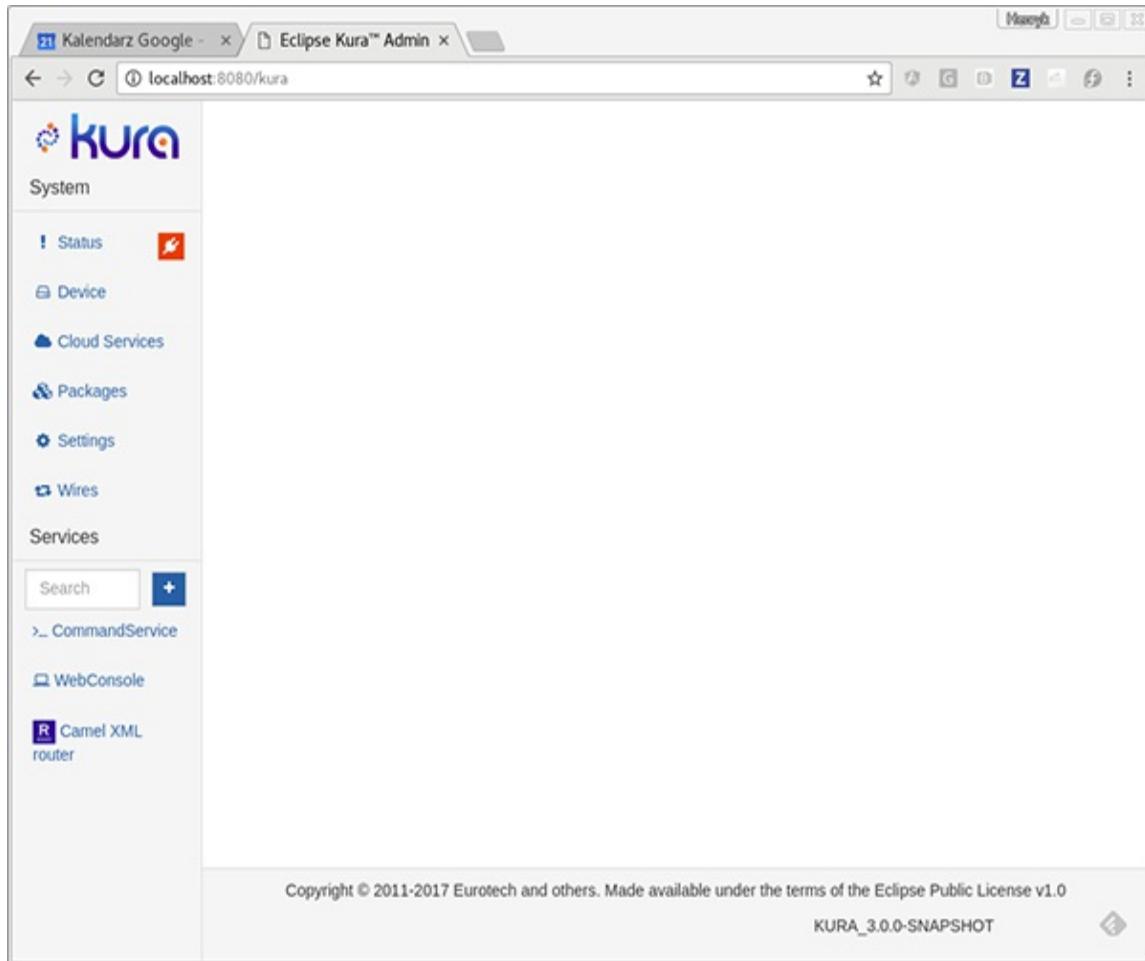


Figure 21.6 Kura web UI

As you can see, Kura provides a bunch of useful features:

- Reviewing information about a given device
- Managing OSGi bundles installed on devices
- Connecting to a back-end services platform (for example, to Eclipse Kapua)

The feature that's the most interesting is hidden under the Camel XML Router tab. Eclipse Kura happens to come with Apache Camel installed by default. Among other things, Camel support allows you to define XML routes using the web UI.

Let's try to create a new Camel XML in a Kura server. To do that, navigate to the Camel XML Router tab. After you click it, you should see a large Router XML tab (shown in [figure 21.7](#)).

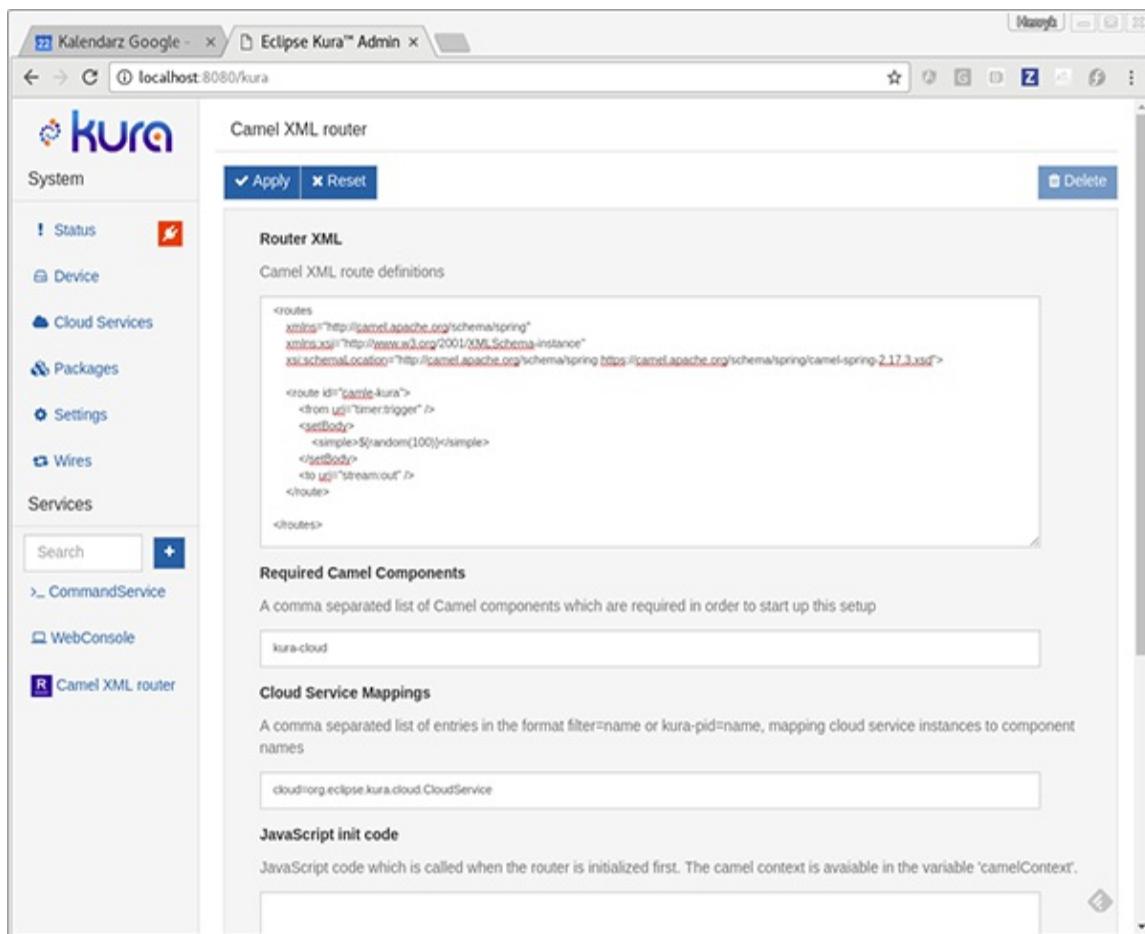


Figure 21.7 Camel routes in the Kura web UI

Now copy the code in the following listing and paste it into the Camel XML input. After that, click the Apply button.

Listing 21.1 Camel XML route to copy into Kura web UI

```
<routes
    xmlns="http://camel.apache.org/schema/spring"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://camel.apache.org/schema/spring
https://camel.apache.org/schema/spring/camel-spring.xsd">

    <route id="camle-kura">
        <from uri="timer:trigger"/>
        <setBody>
            <simple>${random(100)}</simple>
        </setBody>
        <to uri="stream:out"/>
    </route>
</routes>
```

```
</route>  
  
</routes>
```

This code generates a random number from 0 to 99 every second and prints that number into standard output. To see those numbers generated by Camel, navigate to your shell again and execute the following command:

```
docker logs kura
```

As a result, you should see a stream of random numbers similar to the following:

```
3  
12  
79  
28  
71  
4  
38
```

21.5.3 NEXT STEPS WITH CAMEL AND KURA

In the previous section, you managed to start the Eclipse Kura emulator as a Docker container and deploy a Camel route into it. If you're interested in using Camel and Kura together for more serious use cases than a simple Hello World example, you should think about getting your Kura server connected to a back-end IoT platform.

Although Kura can connect to various cloud providers, the primary back-end platform for Kura is the Eclipse Kapua project. Kapua can be used to manage Kura servers connected to it and to provision resources on individual devices. In particular, it's possible to manage OSGi bundles deployed into a particular Kura server. In practice, this means you can use Kapua to manage Camel routes installed into your Kura-enabled device.

21.6 Next steps with Camel and the IoT

In this chapter, you've learned about the Internet of Things and

how to take advantage of Apache Camel to create applications for this domain. You've also learned how Eclipse Kura can be used as a device deployment platform for your Camel routes. If you want to continue your IoT journey from this point, I recommend becoming familiar with the following topics.

ECLIPSE KAPUA

When creating an IoT system, sooner or later you'll discover a need for a back-end platform for your connected devices. Eclipse Kapua is an open source back-end platform supporting Camel as a first-class citizen.

LWM2M AND THE ECLIPSE LESHAN PROJECT

LWM2M is an advanced device management protocol that has a real chance to become a de facto standard for orchestrating a fleet of your connected things. An open source implementation of the LWM2M protocol is called Eclipse Leshan (www.eclipse.org/leshan/).

ECLIPSE HONO

Another IoT project from the Eclipse foundation is called Hono (www.eclipse.org/hono/). Hono, an extension to the AMQP messaging layer, can be used to create a scalable multitenant layer between your devices and your back-end platform. This is a project that's particularly interesting in the context of Camel, as Camel comes with good AMQP connectivity support.

21.7 Summary

This chapter covered the basic concepts of the Internet of Things (IoT) in the context of Camel-based applications. It started with a short shopping list to equip you with basic and cheap, yet powerful, IoT hardware. It also briefly discussed IoT systems architecture and challenges. And this chapter presented scenarios in which Camel brings the most benefits to the IoT messaging infrastructure.

The IoT is an extremely wide topic covered in many books. The aim of this chapter was to give you a taste of how interesting IoT development can be, especially when you're equipped with such a powerful tool as Apache Camel.

I hope you're now convinced that Camel and IoT are a good match and that you'd like to investigate this topic further. I strongly encourage you to follow Eclipse IoT communities such as Eclipse Kura or Eclipse Kapua, which are incubators of innovations related to Camel and the IoT.

Henryk Konsek is a senior software engineer at Red Hat. His areas of expertise include data-intensive back-end systems, data streaming, and messaging solutions. Henryk's primary duty at Red Hat is working on data-streaming back-end services for large-scale IoT deployments. He is a committer to a wide range of open source projects related to messaging and the IoT, including Apache Camel, Eclipse Hono, Eclipse Kapua, and Eclipse Kura.

appendix A

Simple, the expression language

Camel offers a powerful expression language, which was called *Simple* because back in the earlier days, it wasn't very powerful. It's evolved to become much more since then, but don't worry: it's still simple to use. The Simple language is provided out of the box in the camel-core JAR file, which means you don't have to add any JARs on the classpath to use it.

A.1 Introducing Simple

In a nutshell, the Simple expression language evaluates an expression on the current instance of Exchange that's under processing. The Simple language can be used for both expressions and predicates, which makes it perfect to use in your Camel routes.

For example, the Content-Based Router EIP can use the Simple language to define predicates in the `when` clauses, as shown here:

```
from("activemq:queue:quotes")
    .choice()
        .when(simple("${body} contains 'Camel'"))
            .to("activemq:camel")
        .when(simple("${header.amount} > 1000"))
            .to("activemq:bigspender")
        .otherwise()
            .to("activemq:queue:other");
```

The equivalent XML DSL example is as follows:

```
<route>
    <from uri="activemq:queue:quotes"/>
    <choice>
        <when>
```

```

        <simple>${body} contains 'Camel'</simple>
            <to uri="activemq:camel"/>
        </when>
        <when>
            <simple>${header.amount} &gt; 1000</simple>
            <to uri="activemq:bigspender"/>
        </when>
        <otherwise>
            <to uri="activemq:queue:other"/>
        </otherwise>
    </choice>
</route>

```

As you can see, the Simple expression is understandable and similar to other scripting languages. In these examples, Camel will evaluate the expression as a Predicate, which means the result is a boolean, which is either `true` or `false`. In the example, you use operators to determine whether the message body contains the word `camel` or whether the message header `amount` is larger than `1000`. Notice also how you had to escape the `>` in the XML DSL with `>`. This applies to any other special characters in XML, like `<` (replaced with `$lt;`) or `&` (replaced with `&`).

That gives you a taste of the Simple language. Let's look at its syntax.

A.2 Syntax

The Simple language uses `${ }` placeholders for dynamic expressions, such as those in the previous examples. You can use multiple `${ }` placeholders in the same expression, or even nest placeholders.

For example, these are valid Simple expressions:

```
"Hello ${header.name} thanks for ordering ${body}"
"${header.${header.bar}}"
```

An alternative syntax is available to accommodate a clash with Spring's property placeholder feature. You can now also use `$simple{ }` placeholders with Simple, as shown here:

```
"Hello ${simple{header.name}} thanks for ordering  
$simple{body}"
```

These examples use variables such as body and header. The next section covers these variables.

A.3 Built-in variables

The Simple language provides variables that bind to information in the current Exchange. You've already seen body and header. Table A.1 lists all the variables available.

Table A.1 Variables in the Simple language

Variable	Type	Description
body in.body	o b j e c t	Contains the input message body. Note that body is preferred over in.body.
header. xxx headers. xxx in.head er.xxx in.head ers.xxx header[xxx] headers[xxx] in.head er[xxx] in.head ers[xxx]	o b j e c t	Contains the input message header xxx.
exchang eProper	o b	Contains the exchange property xxx.

ty.xxx exchangeProperty[xxx]	ject	
headers in.headers	Map	The input message headers.
exchangeId	String	Contains the unique ID of the Exchange.
sys.xxx sysenv. xxx	String	Contains the system environment variable xxx.
exception	Object	Contains the exception on the Exchange, if any exists.
exception.stacktrace	String	Contains the exception stacktrace on the Exchange. If not set, will fall back to the Exchange.EXCEPTION_CAUGHT property on the exchange.
exception.message	String	Contains the exception message on the Exchange, if any exists.
threadName	String	Contains the name of the current thread; can be used for logging purposes.
camelId	St	The camelContext name.

	r i n g	
exchange	E x c h a n g e	The current Exchange object.
routeID	S t r i n g	The ID of the route where this Exchange is currently being routed.
null		Represents null.
message History	S t r i n g	The message history of the current exchange. This shows how the message has been routed.
message History (false)	S t r i n g	The message history of the current exchange, but without showing any of the Exchange content. Helpful if you don't want sensitive details showing up in the logs.
\n	S t r i n g	A newline character.
\t	S t r i n g	A tab character.
\r	S t r i	A carriage return character.

	n g	
\}	s t r i n g	The } character, which is special for a simple expression, of course.

Notice that all the `in.*` variables from table A.1 are being considered for removal in Camel 3.0. Instead, use the non-`in.*` variables. The variables can easily be used in a Simple expression, as you've already seen. Logging the message body can be done by using `${body}`, as shown in the following route snippet:

```
from("activemq:queue:quotes")
    .log("We received ${body}")
    .to("activemq:queue:process");
```

The Simple language also has a set of built-in functions.

A.4 Built-in functions

The Simple language has many functions at your disposal, as listed in table A.2.

Table A.2 Functions provided in the Simple language

Function	Type	Description
<code>bodyAs(type)</code>	<code>t y p e</code>	Converts the body to the given type. For example, <code>bodyAs(String)</code> or <code>bodyAs(com.foo.MyType)</code> . Returns <code>null</code> if the body can't be converted.
<code>bodyAs(type).OGNL</code>	<code>o b j e c t</code>	Converts the body to the given type and then invokes a method on the resulting object by using Object-Graph Navigation Language (OGNL) notation. May return <code>null</code> if the body can't be converted or if the method returns <code>null</code> .

mandatoryBodyAs(type)	type	Converts the body to the given type. Throws a <code>NoTypeConversionAvailableException</code> if the body can't be converted.
mandatoryBodyAs(type).OGNL	object	Converts the body to the given type and then invokes a method on the resulting object by using OGNL notation. Throws a <code>NoTypeConversionAvailableException</code> if the body can't be converted.
headerAs(key, type)	type	Converts the header with the given key to the given type. Returns <code>null</code> if the header can't be converted.
bean:beanId[?method]	object	Invokes a method on a bean. Camel looks up the bean with the given ID from the <code>Registry</code> and invokes the appropriate method. You can optionally explicitly specify the name of the method to invoke.
date:command:pattern	string	<p>Formats a date. The command must be either <code>now</code> or <code>header.XXX</code>: <code>now</code> represents the current timestamp, whereas <code>header.XXX</code> uses the header with the key <code>xxx</code>.</p> <p>The pattern is based on the <code>java.text.SimpleDateFormat</code> format.</p>
camelContext.ognl	object	Invokes a method on the <code>CamelContext</code> by using OGNL notation. You can see more about OGNL later in this appendix.
collate(sub_list_size)	iterator	Splits a message body into sublists of size <code>sub_list_size</code> . The result is an iterator that points to the sublists.
exchange.ognl	object	Invokes a method on the <code>Exchange</code> by using OGNL notation. You can see more about OGNL later in this appendix.
exchange	o	Gets the exchange property <code>xxx</code> and then invokes a method on

Property .XXX.ogn L	b j e c t	the resulting object by using OGNL notation.
properties-location : [locations:]key	s t r i n g	Resolves a property with the given key by using the Camel Properties component.
properties:key[:default]	s t r i n g	Resolves a property with the given key by using the Camel Properties component. If the key doesn't exist or has no value, a specified default can be used.
random(max)	I n t e g e r	Returns a random number between 0 (included) and the specified max (excluded).
random(min, max)	I n t e g e r	Returns a random number between the specified min (included) and the specified max (excluded).
ref:xxx	o b j e c t	Looks up and returns a bean with ID xxx.
skip(number_of_items)	I t e r a t o r	Skips the specified number of items in the current message body and returns the remainder.
type:name.field	o b j	Refers to a type or a field by its FQN. For example, \${type:org.apache.camel.Exchange.FILE_NAME} refers to the constant Exchange.FILE_NAME field, which resolves to

	e	CamelFileName.
	c	
	t	

Lets try a few of the functions from table [A.2](#). We'll start with the date function. To log a formatted date from the message header, you can do as follows:

```
<route>
    <from uri="activemq:queue:quote"/>
    <log message="Quote date ${date:header.myDate:yyyy-MM-
dd HH:mm:ss}"/>
    <to uri="activemq:queue:process"/>
</route>
```

In this example, the input message is expected to contain a header with the key `myDate`, which should be of type `java.util.Date`(or a long, which will be converted to a `java.util.Date` by Camel automatically).

Suppose you need to organize received messages into a directory structure containing the current day's date as a parent folder. The file producer has direct support for specifying the target filename by using the Simple language as shown in bold:

```
from("activemq:queue:quote")
    .to("file:backup/?fileName=${date:now:yyyy-MM-
dd}/${exchangeId}.txt")
    .to("activemq:queue:process");
```

Now suppose the file must use a filename generated from a bean. You can use the `bean` function to achieve this:

```
from("activemq:queue:quote")
    .to("file:backup/?fileName=${bean:uuidBean?
method=generate}}")
    .to("activemq:queue:process");
```

In this example, Camel looks up the bean with the ID `uuidBean` from the Registry and invokes the `generate` method. The output of this method invocation is returned and used as the filename.

The Camel Properties component is used for property placeholders. For example, you can store a property in a file

containing a configuration for a big-spender threshold:

```
big=5000
```

Then you can refer to the `big` properties key from the Simple language:

```
from("activemq:queue:quotes")
    .choice()
        .when(simple("${header.amount} >
${properties:big}")
            .to("activemq:bigspender")
        .otherwise()
            .to("activemq:queue:other");
```

The Simple language also has built-in variables when working with the Camel File and FTP components.

A.5 Built-in file variables

Files consumed using the File or FTP components have file-related variables available to the Simple language. Table A.3 lists those variables.

Table A.3 File-related variables available when consuming files

Variable	Type	Description
<code>file:name</code>	string	Contains the filename (relative to the starting directory).
<code>file:name.extension</code>	string	Contains the file extension.
<code>file:name.extension.substring</code>	string	Contains the file extension whereby only the extension after the last dot is returned. A tar.gz file would have an extension of gz.

le	in g	
file:n ame.no ext	st r i n g	Contains the filename without extension (relative to the starting directory).
file:n ame.no ext.si ngle	st r i n g	Contains the filename without extension (relative to the starting directory), whereby only the extension after the last dot is stripped. A file named my-dir/backup.tar.gz would mean my-dir/backup.tar is returned.
file:o nlynam e	st r i n g	Contains the filename without any leading paths.
file:o nlynam e.noex t	st r i n g	Contains the filename without extension and leading paths.
file:o nlynam e.noex t.sing le	st r i n g	Contains the filename without extension and leading paths, whereby only the extension after the last dot is stripped. A file named backup.tar.gz would mean backup.tar is returned.
file:p arent	st r i n g	Contains the file parent (the paths leading to the file).
file:p ath	st r i n g	Contains the file path (including leading paths).
file:a bsolut	Bo	Indicates whether the filename is an absolute or relative file path.

e	o l e a n	
file:absolute.path	s t r i n g	Contains the absolute file path.
file:length	l o n g	Contains the file length.
file:size	D a t e	Contains the modification date of the file as a <code>java.util.Date</code> type.

Among other things, the file variables can be used to log which file has been consumed:

```
<route>
    <from uri="file://inbox"/>
    <log message="Picked up ${file:name}"/>
    ...
</route>
```

The File and FTP endpoints have options that accept Simple language expressions. For example, the File consumer can be configured to move processed files into a folder you specify. Suppose you must move files into a directory structure organized by dates. You can do that by specifying the expression in the `move` option, as follows:

```
<from uri="file://inbox?
move=backup/${date:now:yyyyMMdd}/${file:name}">
```

TIP The FTP endpoint supports the same move option as shown here.

Another example where the file variables come in handy is if you have to process files differently based on the file extension. For example, suppose you have CSV and XML files:

```
from("file:///inbox")
    .choice()
        .when(simple("${file:ext} == 'txt')).to("direct:txt")
        .when(simple("${file.ext} == 'xml')).to("direct:xml")
        .otherwise().to("direct:unknown");
```

NOTE You can read more about the file variables at the Camel website: <http://camel.apache.org/file-language.html>.

In this appendix, we've used the Simple language for predicates. In fact, the previous example determines whether the file is a text file. Doing this requires operators.

A.6 Built-in operators

The first example in this appendix implemented the Content-Based Router EIP with the Simple expression language. It used predicates to determine where to route a message, and these predicates use operators. Table A.4 lists all the operators supported in Simple.

Table A.4 Operators provided in the Simple language

Operator	Description
==	Tests whether the left side is equal to the right side
=~	Tests whether the left side is equal to the right side, ignoring case
>	Tests whether the left side is greater than the right side
>=	Tests whether the left side is greater than or equal to the right side

<	Tests whether the left side is less than the right side
<=	Tests whether the left side is less than or equal to the right side
!=	Tests whether the left side isn't equal to the right side
contains	Tests whether the left side contains the string value on the right side
not contains	Tests whether the left side doesn't contain the string value on the right side
starts with	Tests whether the left side starts with the string value on the right side
ends with	Tests whether the left side ends with the string value on the right side
in	Tests whether the left side is in a set of values specified on the right side; the values must be separated by commas
not in	Tests whether the left side isn't in a set of values specified on the right side; the values must be separated by commas
range	Tests whether the left side is within a range of values defined with the following syntax: 'from..to'
not range	Tests whether the left side isn't within a range of values defined with the following syntax: 'from..to'
regex	Tests whether the left side matches a regular expression pattern defined as a string value on the right side
not regex	Tests whether the left side doesn't match a regular expression pattern defined as a string value on the right side
is	Tests whether the left-side type is an instance of the value on the right side
not is	Tests whether the left-side type isn't an instance of the value on the right side

++	Unary operator that increments a left-side function and returns that value. So if you have a counter header with <code>value = 1</code> , the expression <code> \${header.counter}++</code> would return 2.
--	Unary operator that decrements a left-side function and returns that value. If you have a counter header with <code>value = 2</code> , the expression <code> \${header.counter}--</code> would return 1.

The operators require the following syntax:

```
 ${leftValue} <OP> rightValue
```

The value on the left side must be enclosed in a `${ }` placeholder. The operator must be separated with a single space on the left and right. The right value can either be a fixed value or another dynamic value enclosed using `${ }` .

Let's look at an example:

```
simple("${in.header.foo} == Camel")
```

Here you test whether the `foo` header is equal to the string value "Camel". If you want to test for "Camel rocks", you must enclose the string in quotes (because the value contains a space):

```
simple("${in.header.foo} == 'Camel rocks'")
```

Camel automatically type coerces, so you can compare apples to oranges. Camel will regard both as fruit:

```
simple("${in.header.bar} < 200")
```

Suppose the `bar` header is a string with the value "100". Camel will convert this value to the same type as the value on the right side, which is numeric. It will therefore compute `100 < 200`, which renders true.

You can use the range operator to test whether a value is in a numeric range.

```
simple("${in.header.bar} range '100..199'")
```

Both the *from* and *to* range values are inclusive. You must define

the range exactly as shown.

A regular expression can be used to test a variety of things, such as whether a value is a four-digit value:

```
simple("${in.header.bar} regex '\d{4}'")
```

You can also use the built-in functions with the operators. For example, to test whether a given header has today's date, you can use the date function:

```
simple("${in.header.myDate} == ${date:now:yyyyMMdd}")
```

TIP You can see more examples in the Camel Simple online documentation: <http://camel.apache.org/simple.html>.

The Simple language also allows you to combine two expressions.

A.6.1 COMBINING EXPRESSIONS

The Simple language can combine expressions via the `&&` (and) or `||` (or) operators. The syntax for combining two expressions is as follows:

```
 ${leftValue} <OP> rightValue && or ||> ${leftValue} <OP>  
rightValue
```

Here's an example using `&&` to group two expressions:

```
simple("${in.header.bar} < 200 && ${body} contains  
'Camel'")
```

The Simple language also supports an OGNL feature.

A.7 The OGNL feature

Both the Simple language and Bean component support an Object-Graph Navigation Language (OGNL) feature when specifying the method name to invoke. OGNL allows you to

specify a chain of methods in the expression.

Suppose the message body contains a `Customer` object that has a `getAddress` method. To get the ZIP code of the address, you type the following:

```
simple("${body.getAddress().getZip()}")
```

You can use a shorter notation, omitting the `get` prefix and the parentheses:

```
simple("${body.address.zip}")
```

In this example, the ZIP code will be returned. But if the `getAddress` method returns `null`, the example would cause a `NoSuchMethodException` to be thrown by Camel. If you want to avoid this, you can use the null-safe operator `?.` as follows:

```
simple("${body?.address.zip}")
```

The methods in the OGNL expression can be any method name. For example, to invoke a `sayHello` method, you do this:

```
simple("${body.sayHello}")
```

Camel uses the bean parameter binding (covered in chapter 4). This means that the method signature of `sayHello` can have parameters that are bound to the current `Exchange` being routed:

```
public String sayHello(String body) {  
    return "Hello " + body;  
}
```

The OGNL feature has specialized support for accessing `Map` and `List` types. For example, suppose the `getAddress` method has a `getLines` method that returns a `List`. You could access the lines by their index values, as follows:

```
simple("${body.address.lines[0]}")  
simple("${body.address.lines[1]}")  
simple("${body.address.lines[2]}")
```

If you try to index an element that's out of bounds, an

`IndexOutOfBoundsException` exception is thrown. You can use the null-safe operator to suppress this exception:

```
simple("${body.address?.lines[2]}")
```

If you want to access the last element, you can use `last` as the index value, as shown here:

```
simple("${body.address.lines[last]}")
```

The access support for `Maps` is similar, but you use a key instead of a numeric value as the index. Suppose the message body contains a `getType` method that returns a `Map` instance. You could access the `gold` entry as follows:

```
simple("${body.type[gold]}")
```

You could even invoke a method on the `gold` entry like this:

```
simple("${body.type[gold].sayHello}")
```

This concludes our tour of the various features supported by the Camel Simple language. We'll now take a quick look at how to use the Simple language from custom Java code.

A.8 Using Simple from custom Java code

The Simple language is most often used directly in your Camel routes, in either the Java DSL or XML DSL file. But it's also possible to use it from custom Java code.

Here's an example that uses the Simple language from a Camel Processor.

[Listing A.1](#) Using the Simple language from custom Java code

```
package camelinaction;

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

```

public class MyProcessor implements Processor {
    public void process(Exchange exchange) throws Exception
{
    SimpleBuilder simple = new SimpleBuilder(
        "${body} contains
'Camel'");
        if (!simple.matches(exchange)) {
            throw new Exception("This is NOT a Camel
message");
        }
}

```

As you can see, all it takes is creating an instance of `SimpleBuilder`, which is capable of evaluating either a predicate or an expression. In the listing, you use the Simple language as a predicate.

To use an expression to say “Hello”, you could do the following:

```

SimpleBuilder simple = new SimpleBuilder("Hello
${header.name}");
String s = simple.evaluate(exchange, String.class);
System.out.println(s);

```

Notice how you specify that you want the response back as a `String` by passing in `String.class` to the `evaluate` method.

Listing A.1 uses the Simple language from within a Camel Processor, but you’re free to use it anywhere, such as from a custom bean. Just keep in mind that the `Exchange` must be passed into the `matches` method on the `SimpleBuilder`.

Summary

This appendix covered the Simple language, an expression language provided with Camel. You saw how well it blends with Camel routes, which makes it easy to define predicates in routes, such as those needed when using the Content-Based Router.

We also looked at how easy it is to access information from the

Exchange message by using Simple's built-in variables. You saw that Simple provides functions, such as a date function that formats dates and a bean function that invokes methods on beans.

Finally, we covered OGNL notation, which makes it even easier to access data from nested beans.

The Simple language is a great expression language that should help you with 95 percent of your use cases.

appendix B

The Camel community

As with any open source project, the community behind Camel is extremely important. We think of *community* as an all-encompassing term for the official website, mailing lists, the issue tracker, Camel users, projects based on or extending Camel, and much more. Measuring the vibrancy of a community when it has this many moving parts can be hard, but it's important. Having a stagnant or small community will make your (the user's) experience more difficult when things go wrong and during development in general. Camel's community is highly active and expanding, so you're in luck!

This appendix covers main aspects of the Camel community.

Apache Camel website

The Apache Camel website, <http://camel.apache.org>, will be your main resource when using Camel. You may have already noticed that we reference pages from the Camel website throughout the book. It's a good resource for us too. On the Camel website, you'll find links for downloads, documentation, support, and many other topics.

JIRA, mailing lists, Gitter, and IRC

When things don't go as planned, you'll need to get help from people in the Camel community. If you know you have a problem with a demonstrative test case or have a feature request, you can create a ticket in Camel's JIRA instance: <https://issues.apache.org/jira/browse/CAMEL>. From there, one of the Camel developers will take the ticket and possibly commit

a fix for the issue. If you want to fix the issue yourself, add a comment to the JIRA issue and then use GitHub to contribute the fix, as explained in the next section. More information on contributing is also available at <http://camel.apache.org/contributing.html>.

When you have general questions, you can send an email to the Camel user mailing list (<http://camel.apache.org/mailings-lists.html>). Your question will be answered by one of the Camel developers or another Camel user.

You can even chat in real time with a Camel developer or user on the Camel Internet Relay Chat (IRC) chat room (<http://camel.apache.org/irc-room.html>) or Gitter channel at <https://gitter.im/apache/camel>.

Camel at GitHub

GitHub has revolutionized open source collaboration. It's hard these days to find an active open source project that hasn't moved over to GitHub! Camel can be found on GitHub at <https://github.com/apache/camel>. For folks wanting to submit a potential bug fix, submit a fix for docs, or contribute a new example, GitHub pull requests (PRs) are an easy way for Camel maintainers to review, provide feedback about, and merge your code. PRs are the preferred method of code contribution.

While you check out Camel at GitHub, it's also nice to star the project to make the maintainers feel appreciated. ;-)

Camel at Stack Overflow

These days, Stack Overflow is the top question-and-answer website for all things related to programming. Many people use Stack Overflow instead of posting questions to the official Apache Camel mailing list. This is fine and can be considered part of the wider Camel community. Committers frequently search out and help folks posting questions on Stack Overflow.

Be sure when asking or answering a Camel-related topic to use the tag *apache-camel* so it's noticed. At the time of writing, more than 7,000 topics have been discussed:
<https://stackoverflow.com/questions/tagged/apache-camel>.

Commercial Camel offerings

Plenty of companies specialize in Camel and offer paid support as well as consulting. The full list is at
<http://camel.apache.org/commercial-camel-offerings.html>.

One company on that list is Red Hat, where both authors are employed. Apache Camel support at Red Hat comes from the JBoss Fuse product line, which is an open source integration platform. Red Hat does the same for other popular Apache projects as well, including ActiveMQ, Artemis, CXF, Karaf, ServiceMix, Tomcat, and Qpid. The JBoss Fuse team includes founders, PMC members, and many of the committers to Apache Camel, and they know the code better than anyone else does.¹

The JBoss Fuse website (<https://developers.redhat.com/products/fuse/overview/>) includes free downloads, documentation, training videos, webinars, and other tools to help developers get started and be successful with JBoss Fuse and Apache Camel.

¹ What else would you expect us to say about our coworkers? They rock. :-)

Camel tooling

Since the first edition of this book, tooling for Camel has expanded such that we've had to devote an entire chapter to this topic. None of the GUI tooling is developed at the Apache Camel project, so it can be considered part of the wider community. Check out tooling projects such as hawtio, JBoss Fuse Tooling, and the Camel plugin for IntelliJ IDEA in chapter 19.

Camel-extra project

Camel ships with many components, but other components are available separately from the camel-extra project at GitHub (<https://camel-extra.github.io>). The main motivator for not including all Camel components in the main distribution is licensing. Apache Camel is developed and distributed under the Apache License, version 2. The camel-extra project contains components that integrate with libraries that have GPL and LGPL licenses, which are disallowed by the Apache Software Foundation.

At camel-extra, you'll find components for integrating with the following, among others:

- The Esper Event Stream Processing library
- The IBM Customer Information Control System (CICS)
- CIFS/SMB networking protocol
- The Hibernate ORM tool

The components from camel-extra aren't officially affiliated with or supported by Apache. In addition, the project doesn't always follow Camel's release cycle, so is more likely to be outdated.

Becoming a Camel committer

Yes, it's possible to become an Apache Camel committer yourself. To do this, you must contribute to the community. Contributions could be a combination of answering questions on the mailing list or Stack Overflow, contributing new features or bug fixes via PRs, or writing blogs or articles about Camel. The key is to keep up with contributions over a period of time. There isn't a set period of time or amount of code you need to submit; it varies. But after a while, someone from the Camel Project Management Committee (PMC) will invite you to become a committer on the Camel project. After you accept, you'll be able to commit code directly to Camel yourself.

Videos

There have been many presentations on Apache Camel over the years. Some of them were video recorded and are available online on popular sites such as YouTube or Vimeo. You may also find video recordings from the Camel team showing some cool new functionality that are on the way.

Other resources

The Camel website has an extensive collection of links to external articles, blogs, books (<http://camel.apache.org/books.html>—there are nine books out now!), projects, presentations, videos, podcasts, and other sources that cover Camel (<http://camel.apache.org/articles.html>). There's also a collection of links to other third-party Camel projects and companies that use Camel (<http://camel.apache.org/user-stories.html>). Dozens of projects and additional Camel components are listed here, so it's worth a look.

If you've written a blog entry or article, or your company uses Camel and wants to have a link added, please contact the Camel team.

Index

Symbols

@Api annotation [447](#)
@ApiModel annotation [448](#)
@ApiOperation annotation [447](#)
@ApplicationPath annotation [271](#)
@ApplicationScoped annotation [267, 272, 299](#)
@ArquillianResource annotation [399](#)
@Attachments annotation [132](#)
@Autowired annotation [109](#)
@Bean annotation [133, 134](#)
@Body annotation [130, 132](#)
@CamelAware annotation [668](#)
@Component annotation [47, 260, 353](#)
@ComponentScan annotation [353](#)
@ConfigurationProperties annotation [265](#)
@Constant annotation [133](#)
@ContextConfiguration annotation [352, 356](#)
@Converter annotation [102, 104](#)
@CsvRecord annotation [95](#)
@DataField annotation [95](#)
@DynamicRouter annotation [182–184](#)

@EnableCircuitBreaker annotation [295](#), [301](#)
@EnableDiscoveryClient annotation [749](#)
@Entity[Entity] annotation [222](#)
@ExchangeException annotation [132](#)
@ExchangeProperties annotation [132](#)
@Groovy annotation [133](#)
@Handler annotation [123](#), [124](#)
@Header annotation [132](#), [267](#)
@Headers annotation [132](#), [480](#)
@HystrixCommand annotation [293](#), [294](#), [303](#)
@Ignore annotation [336](#)
@Inject annotation [118](#), [249–250](#)
@JavaScript annotation [133](#)
@JsonPath annotations [135–136](#)
@ManagedAttribute annotation [699](#), [709](#), [711](#)
@ManagedOperation annotation [711](#)
@ManagedResource annotation [709](#), [711](#)
@MVEL annotation [133](#)
@Named annotation [118](#), [267](#), [795](#)
@NamespacePrefix annotation [135](#)
@Observes annotation [251](#)
@Path annotation [412](#)
@PathParam annotation [412](#)
@PortForward annotation [795](#)
@PropertyInject annotation [249](#)
@Recipient-List annotation [67](#)

@RequestMapping annotation [258](#), [265](#)
@RequiresKubernetes annotation [795](#)
@RestController annotation [258](#), [265](#)
@RoutingSlip annotation [180](#)–[182](#)
@RunWith annotation [351](#)–[352](#), [354](#)–[355](#)
@Service annotation [293](#)
@Simple annotation [133](#)
@Singleton annotation [118](#), [247](#)
@SpEL annotation [133](#)
@SpringBootApplication annotation [356](#), [749](#)
@SpringBootTest annotation [356](#), [357](#)
@Test annotation [347](#)
@Uri annotation [250](#)–[251](#)
@WebServlet annotation [365](#)
@XmlAccessorType [90](#)
@XmlAttribute annotation [90](#)
@XmlRootElement annotation [90](#), [440](#)
@XPath annotation [68](#), [133](#), [135](#), [517](#)
@XQuery annotation [133](#)
\${ } placeholders [50](#), [57](#), [824](#)
& (ampersand character) [44](#)
200 status code, Rider Auto Parts [457](#)
31068 port [766](#)
400 status code, Rider Auto Parts [457](#)
404 status code, Rider Auto Parts [457](#)
5005 port [774](#)

500 status code, Rider Auto Parts [457](#)

5432 port [738](#)

5701 port [723](#)

5702 port [723](#)

8080 port [766](#)

8500 port [723](#)

. (period) characters [212](#)

+ (plus) characters [233](#)

+ signs [175](#)

A

access attribute [619](#)

Access-Control-Allow-Headers [462](#)

Access-Control-Allow-Methods header [462](#)

Access-Control-Allow-Origin header [462](#)

Access-Control-Max-Age header [462](#)

accessDecisionManager attribute [619](#)

AccessDecisionManager bean [619](#)

accessing

 data with JDBC components [218–221](#)

 remote files with FTP components [203–204](#)

ACID (atomic, consistent, isolated, and durable) [515, 530, 533](#)

acknowledge method [521](#)

action entry [601](#)

active/active mode

 active/passive mode vs. [725–726](#)

clustered [720–721](#)

active/passive mode

- active/active mode vs. [725–726](#)
- clustered [720](#)
 - using Consul [723–724](#)
 - using Hazelcast [721–723](#)
 - using Zookeeper [724–726](#)

ActiveMQ [4, 32](#)

- configuring broker-level redelivery with [538](#)
- configuring consumer-level redelivery with [537](#)

activemq:myActiveMQQueue [47](#)

ActiveMQ broker [519](#)

ActiveMQ component [205](#)

ActiveMQConnectionFactory [32, 205](#)

ActiveMQ message broker [726–727](#)

ActiveMQ-XAConnectionFactory [532](#)

activity logging [572](#)

addAsLibraries method [367](#)

addPackages method [367](#)

addRoutes method [34](#)

addServicesOnStartup method [361](#)

addSynchronization method [545, 547](#)

admin bean [618](#)

Advanced Message Queuing Protocol. *See* AMQP (Advanced Message Queuing Protocol)

adviceWith method [382, 383, 521](#)

adding interceptors to existing routes with [382–383](#)
amending routes using weave with [387–390](#)
 selecting nodes with weave [389–390](#)
 weave without using IDs [388–389](#)
manipulating routes with [383–387](#)
 advising routes [384–385](#)–
 replacing route endpoints [385](#)–
AdviceWithRouteBuilder [385](#)
advising routes [384–385](#)
AffirmativeBased [620](#)
afterConfigureProperties method [336](#)
aggregatedData bean [582](#)
aggregated exchange properties [155–156](#)
aggregate method [85](#)
<aggregate> tag [150](#)
aggregating split messages [174–175](#)
aggregationRepositoryRef attribute [161](#)
AggregationStrategy
 example [151](#)
 handing exceptions with [176–178](#)
 thread safe [152](#)
 using [152](#)
 using POJO for [158–159](#)
AggregationStrategy class [86](#), [149](#), [150](#), [151–152](#)
Aggregator EIPs (enterprise integration patterns)
 aggregated size [156](#)

AggregationStrategy [175](#)
aggregation strategy [149](#), [150](#)
combining message [151](#), [152](#), [175](#)
 AggregateController completion [153](#)
 Batch Consumer completion [153](#)
 conditions provided [153](#)
 examples [154](#)
 Interval completion [153](#)
 OnStop completion [153](#)
 predicate completion [157](#)
 Predicate completion [153](#)
 size completion [154](#)
 Size completion [153](#)
 timeout completion [152](#)
 Timeout completion [153](#)
 using multiple conditions [156](#)
completion conditions for [150](#), [152–159](#)
 aggregated exchange properties [155–156](#)
 configuration options [156](#)
 using multiple completion conditions [154–155](#)
 using POJO for AggregationStrategy [158–159](#)
completion size [150](#)
 closeCorrelationKeyOnCompletion [157](#)
 eagerCheckCompletion [157](#)
 groupExchanges [157](#)
 mandatory [148](#)

correlation expression [150](#)
correlation group [152](#)
correlation identifier [149](#), [152](#), [156](#)
 invalid [157](#)
ordering [152](#)
overview [147](#)
persistence [163](#)
 AggregationRepository [159](#), [161](#)
 memory repository [159](#)
 pluggable repository [159](#)
 RecoverableAggregationRepository [159](#), [160](#)
recovery [159–166](#)
 background task [164](#), [166](#)
 commit [164](#)
 dead letter channel [166](#)
 HawtDBAggregationRepository [165](#)
 interval [165](#), [166](#)
 maximum redeliveries [165–166](#)
 recover [164](#)
 RecoverableAggregationRepository [166](#)
 redelivered [166](#)
 setup [165](#)
 similar to JMS Broker [163](#)
 transaction [163–165](#)
 using error handler [163](#)
reject incoming message [157](#)

uses cases [148](#)
using AggregationStrategy [151–152](#)
using persistence with [159–161](#), [163](#)
using recovery with [163–165](#), [166](#)

algorithms
(-a) option [597](#)
for method-selection [121–123](#)

aliases [332](#)

allMessages() method [373](#)

allowNull option [104](#)

AllowUseOriginalMessage option [629](#)

ambassador pattern [779](#)

AmbiguousMethodCallException [123](#), [124](#), [125](#), [126](#), [127](#)

amending routes, with adviceWith method [387–390](#)
selecting nodes with weave [389–390](#)
weave without using IDs [388–389](#)

ampersand character (&) [44](#)

AMQP (Advanced Message Queuing Protocol) [244](#)

AngularJS [821](#)

AnnotatedRecipientList class [68](#)

AnnotationConfigApplicationContext [354](#)

annotations
binding parameters with [130–132](#)
in @DynamicRouter [182–184](#)
in @RoutingSlip [180–182](#)
in recipient lists [67–69](#)

using @JsonPath binding annotation [135–136](#)

AnnotationTypeConverterLoader [101](#)

anotherBrickInTheWall method [512](#)

Apache ActiveMQ [47](#)

Apache Aries [532](#)

Apache Camel, website community [835](#)

Apache Camel IDEA tooling [810](#)

Apache CXF

- adding Swagger to applications [448–451](#)
- beans [422–424](#)
- HTTP transport [652](#)
- link to website [599](#)

Apache Derby [518](#)

Apache Felix [658](#)

Apache FreeMarker [99](#)

Apache Geronimo [532](#)

Apache Karaf [529, 534, 604, 605, 606, 645, 658, 660](#)

Apache Karaf container, using Jolokia with [682](#)

Apache Maven

- archetype plugin [307–308](#)
- artifactId [308](#)
- central repository [14, 311](#)
- convention over configuration [308](#)
- data directory [13](#)
- dependency tree command [312](#)
- directories installed [13](#)

groupId [308](#)
local download cache [14](#)
obtaining [13](#)
POM [13, 307](#)
pom.xml [13, 308](#)
quickstart archetype [307](#)
src directory [13](#)
transitive dependencies [311](#)

Apache Mesos [244](#)

Apache OpenJPA [224](#)

Apache OpenWebBeans [117](#)

Apache ServiceMix [4, 629, 658](#)

Apache Shiro [618](#)

Apache Tomcat

- deploying to [650–652](#)
- hot deployment [650–653](#)
- leverage servlet transport [651](#)
- using CXFServlet [652](#)
- using for HTTP inbound endpoints [652–654](#)

Apache TomEE [117, 252](#)

Apache Velocity, transforming templates with [99](#)

API (application program interface)

- component frameworks [326–337](#)
- configuring documentation [459–461](#)
- documentation using Swagger [444–465](#)
- documenting error codes [454–457, 459](#)

documenting input [454–459](#)
documenting output [454–459](#)
documenting Rest DSL services [452–453](#)
Swagger overview [445–446](#)
Swagger with JAX-RS REST services [446–451](#)
Swagger with Rest DSL [451–452](#)
using CORS [461–465](#)
using Swagger web console [461–465](#)
generating projects [326–328](#)

API component framework [326](#)

apiComponent option [459](#)
apiContextIdPattern option [459, 460](#)
apiContextListing option [459, 460](#)
apiContextPath option [451, 452, 459](#)
apiContextRouteId option [459, 460](#)
apiDoc attribute [460](#)

API documentation

enabling [459–460](#)
filtering CamelContext in [460–461](#)
filtering routes in [460–461](#)

ApiModelProperty [267](#)

apiProperty [460](#)
application.properties file [260, 261, 264, 265, 300, 749, 755, 760, 791](#)

ApplicationContext [19, 40, 647](#)
applicationContext.xml file [649](#)

ApplicationContextRegistry [114](#), [116](#)

Application level health checks [672](#)

applications

CDI, configuring [248–249](#)

checking health at level of [675–676](#)

creating from beans using Spring [40–43](#)

CXF, adding Swagger to [448–451](#)

debugging in Kubernetes [774–776](#)

running in Kubernetes using Maven tooling [768–770](#)

JAX-RS, using Camel in [414–417](#)

managing [694–698](#)

managing lifecycles of [694–695](#)

using hawtio [695–697](#)

using Jolokia [695–697](#)

migrating to OSGI bundles [400–401](#)

processing messages [392](#)

running using kubectl [765–766](#)

Spring Boot, adding REST to [257–258](#)

testing with Arquillian [395–399](#), [405–](#)

testing with Awaitility [404](#)

testing with Byteman [404](#)

testing with Citrus integration testing [404](#)

testing with Gatling [405](#)

testing with JMC [405](#)

testing with JMH [405](#)

testing with Pax Exam [399–404](#)

running tests [404](#)
writing integration test classes [402–403](#)
testing with Wireshark [405](#)
testing with YourKit [405](#)
tracking activity [684–694](#)
 using core logs [685](#)
 using custom logging [685–689](#)
 using log files [685](#)
 using notifications [691–694](#)
 using Tracer [689–691](#)
application servers, Arquillian [398](#)
archetypeArtifactId property [308](#)
archetypes, Maven [307–311](#)
architecture
 modular [8](#)
 of Camel [18–24](#)
 concepts [19–24](#)
 overview [18](#)
 pluggable [8](#)
arquillian-bom [398](#)
Arquillian Cube extension
 running tests on Kubernetes [793–794](#)
 setting up [792](#)
 writing unit tests using [793](#)
arquillian-junit-container [366, 398](#)
Arquillian testing platform

adding artifacts [398](#)
choosing application servers [398](#)
setting up [397](#)
testing applications with [395–399](#), [405–406](#)
testing WildFly with [366–367](#)

artifacts

- Arquillian, adding [398](#)
- Pax Exam, adding [402](#)

assertEquals method [377](#)
assertIsSatisfied method [370](#), [371](#), [373](#)
assertMockEndpointsSatisfied method [373](#)

asymmetric cryptography [612](#)
AsyncCallback [590](#), [591](#), [593](#)
asynchronous messaging [195](#), [228–230](#)
asynchronous receiver [24](#)
asynchronous routing engine [586–593](#)

- asynchronous API [590–591](#)
- components supporting asynchronous processing [589](#)
- hitting scalability limit [586–587](#)
- potential issues [593](#)
- scalability in Camel [588–589](#)
- writing custom components [591–593](#)

asynchronous throttling [578](#)
AsyncProcessor [590–591](#)
atomic, consistent, isolated, and durable. *See* ACID (atomic, consistent, isolated, and durable)

atomicvalue [735](#)

Atomikos [532](#)

attachments, in messages [16](#)

auditing [69](#)

AuditService bean [686](#), [687](#)

authentication

route authentication and authorization [618–621](#)

web service security [600–604](#)

client-side WS-Security configuration [603–604](#)

server-side WS-Security processing [601–602](#)

using JAAS [604–607](#)

AuthenticationManager bean [619](#)

AuthorizationException [502](#)

auto-acknowledge mode [521](#)

autocaching mode [524](#)

autocomplete [35](#)

autodiscovering

components [196–197](#)

overview [19](#)

autogenerating files [767](#)

automating tasks [195](#)

automocking endpoints [385](#)

auto mode [434](#)

AutoStartup, disabling [634–635](#)

Awaitility DSL, testing applications with [404](#)

AWS-SES component, email [234](#)

B

back ends, for inventory [270–271](#)
backoffErrorThreshold option [509, 538](#)
backoffIdleThreshold option [509, 539](#)
BackOffMultiplier option [483, 509, 538](#)
badOrders queue [59, 60, 61](#)
bar method [124](#)
BarTest class [357](#)
BatchConsumer [153](#)
bean-binding [138](#)
Bean component [198](#)
BeanManager [19](#)
beans [106–145](#)
 as expressions in routes [138–141](#)
 as predicates in routes [138–139, 141–145](#)
 computing routing slip headers with [179–180](#)
 CXF-configured [422–424](#)
 invoking bean defined in XML DSL [108–109](#)
 invoking from Java [108](#)
 method call expression [180](#)
 performing parameter binding [128–138](#)
 using annotations [130–132](#)
 using built-in types [130](#)
 using language annotations [132–136](#)
 using method name with signatures [136–138](#)
 with multiple parameters [129–130](#)

registry 54
selecting methods 119–120, 121–128
method-selection algorithms 121–123
method-selection examples 123–125
method-selection problems 125–127
method selection using type matching 127–128
Service Activator patterns 111–112
to transform data 80–81
using 107–109, 111
using for splitting 170–172
using registries 113–119
ApplicationContextRegistry 116
BlueprintContainerRegistry 116–117
CdiBeanRegistry 117–119
JndiRegistry 114–115
OsgiServiceRegistry 116–117
SimpleRegistry 115–116
using Spring to create applications from 40–43
using to instead of 111
BeforeConsumer mode 550–551
to/from POJOs using camel-jackson 419–420
with Rest DSL 436–439–441
binding
negotiation for messages 435–436
parameters 128–138
multiple 129–130

- using annotations [130–132](#)
- using built-in types [130](#)
- using language annotations [132–136](#)
- using method names with signatures [136–138](#)
- Rest DSL [436](#)
 - to/from POJOs using camel-jaxb [419](#)
 - with Rest DSL [439–441](#)
- bindingMode option [436](#)
- Bindy
 - BindimeType.Csv [96](#)
 - data formats, using [94–97](#)
 - data types used [95](#)
 - specifying package name [96](#)
 - using FIX [97](#)
 - using fixed length [97](#)
- BlockingQueue type [572](#)
- Block option [227](#)
- Blueprint, using Blueprint-based Camel route [661](#)
- BlueprintContainerRegistry [114](#), [116–117](#)
- BlueprintFirstTest class [360](#)
- Blueprint XML, testing with [358–361](#)
- body
 - in messages [16](#)
 - mapping [210–212](#)
- BOM (bill of materials) [252](#)
- Boolean value [138](#)

Bootstrap [821](#)
BPEL (Business Process Execution Language) [244](#)
BPMN (Business Process Model and Notation) [244](#)

 tag [81](#)
bridging consumers, with error handlers [505–509](#)
brokers [31](#)
brokerURL option [519, 537](#)
BrowsableEndpoint [392](#)
Browse component [198](#)
Builder pattern [82](#)
bulkhead patterns [289–292](#)
 fallback via network [292](#)
 fallback with Hystrix [290–291](#)
 time-outs with Hystrix [289–290–291](#)
Burgazolli, Luca [751](#)
Byteman tool, testing applications with [404](#)
BytesMessage [211](#)

C

cache eviction [556](#)
cacheLevelName option [525](#)
caches, clustering [733–737](#)
 using Hazelcast [733–735](#)
 using Infinispan [736–737](#)
 using JCache [736–737](#)
CallbackHandler [602, 604, 607](#)

callbacks, synchronization [546–547](#)

CallerRuns option [573](#)

calling

- commands from Java [283](#)
- microservices [262–276](#)
 - inventory prototype [270–273](#)
 - overview [275–276](#)
 - rating prototype [273–274](#)
 - recommendation prototype [264–266](#)
 - rules prototype [270–273](#)
 - shopping cart prototype [266–269](#)
 - technologies [263–264](#)
 - with fault-tolerance [292–298](#)
- RESTful web services with Rest DSL [443–444](#)

Camel [625–670](#)

CAMEL_ENCRYPTION_PASSWORD variable [599](#)

- archetypes, list of archetypes [308](#)
- architecture of [18–24](#)
- concepts [19–24](#)
- overview [18](#)
- as microservice [245–247](#)
- overview [247](#)
- running Camel standalone using Camel main classes [246](#)
- benefits of using [5–10](#)
 - automatic type converters [9](#)
 - cloud ready [9](#)

community [10](#)

DSLs [7–8](#)

ease of configuration [8](#)

enterprise integration patterns [7](#)

extensive component libraries [6](#)

lightweight core for microservices [9](#)

mediation engines [5](#)

modular architecture [8](#)

payload-agnostic routers [8](#)

pluggable architecture [8](#)

POJOs [8](#)

routing engines [5](#)

Test Kit [9](#)

bridging routes together [228](#)

CDI and [666–669](#)

configuring Netflix Hystrix with [287–288](#)

defined [4–5](#)

deploying [645–658](#)

- embedded in Java application [645–648](#)
- embedded in web application [648–654](#)
- embedded in WildFly [654–658](#)

getting started [10–11, 15](#)

higher-level abstractions [4](#)

installing [10–11](#)

introduction to [4–10](#)

Camel

Java DSL

- choice method [56](#)
- end method [60](#), [63](#)[7](#)
- endsWith predicate [56](#)
- from method [35](#)
- header method [56](#)
- multicast method [63](#)
- noStreamCaching method [63](#)[0](#)
- otherwise method [59](#)
- recipientList method [66](#)
- regex predicate [59](#)
- stop method [60](#)
- stopOnException method [64](#)
- tokenize method [66](#)
- to method [37](#)
- wireTap method [70](#)
- message model [15–18](#)
 - exchange [16–18](#)
 - messages [15–16](#)
- monitoring with Spring Boot [262](#)
- monitoring with WildFly Swarm [255–256](#)
- OSGi and [658–665](#)
 - deploying example [661–662](#)
 - installing and running Apache Karaf [660](#)
 - setting up Maven to generate OSGi bundle [659](#)
 - using Blueprint-based Camel route [661](#)

using MSF to spin up route instances [663–665](#)
predicates and expressions [56](#)
 regular expression predicate [58](#)
 xpath expression [62](#)
protocol support [4](#)
route startup order [628](#)
shutting down [641–645](#)
 considerations regarding [645](#)
 shutting down application [643–645](#)
Spring DSL
 contextScan element [46](#)
 packageScan element [46](#)
 routeContext element [49](#)
 routerBuilder element [46](#)
 when method [56](#)
starting [626–635](#)
 disabling AutoStartup [634–635](#)
 options for [629–631](#)
 ordering routes [631–634](#)
 process for [626–628](#)
starting and stopping routes at runtime [635–641](#)
 using CamelContext [635–638](#)
 using Control Bus EIP [638](#)
 using RoutePolicy [639–641](#)
startup options [630](#)
TypeConverter [38](#)

camel-ahc component [614](#)
camel-api-component-maven-plugins [328–330](#)
camel-apns component [614](#)
camel-archetype-api-component [308](#)
camel-archetype-blueprint [308](#)
camel-archetype-cdi [308](#)
camel-archetype-component [308](#)
camel-archetype-connector [308](#)
camel-archetype-dataformat archetype [308, 338](#)
camel-archetype-java archetype [308–311](#)
camel-archetype-spring-boot [308](#)
camel-archetype-web [308](#)
CamelAuthentication [620](#)
CamelAuthorizationException [621](#)
CamelAware annotation [667](#)
CamelBeanMethodName [121](#)
camel-bindy component [94](#)
CamelBlueprintTestSupport class [358, 360](#)
camel-box component [337, 614](#)
camel-braintree component [337](#)
Camel Catalog [813–816, 822](#)
camel-cdi component [666, 668](#)
CamelCdiRunner [250, 362](#)
camel-cometd component [614](#)
camel-consul component [721](#)
CamelContext

accessing via Processor [78](#)
filtering in API documentation [460–461](#)
namespace [627](#)
preparing in Spring container [627](#)
starting [627–628](#)
starting and stopping routes at runtime [635–638](#)
start method [626](#)
startRoute [635](#)
stopRoute [635](#)
CamelContextAware [196, 704](#)
CamelContext containers [19](#)
camelContext element [44, 45, 49, 51, 53](#)
CamelContextFactoryBean [627](#)
CamelContext parameter [130](#)
camelContext tag [116, 489, 576, 627, 650](#)
Camel-core, adding type converters to [104–105](#)
camel-core module, components [197](#)
camel-core-starter dependency [298](#)
camel-crypto
 encrypting and decrypting payloads using [613–614](#)
 signing and verifying messages using [610–612](#)
camel-cxf [420–425](#)
 adding JSON support to [424–425](#)
 Camel-configured endpoints [420–422](#)
 CXF-configured beans [422–424](#)
camel-cxf.xml file [650, 652](#)

camel-cxf component [414](#), [417](#), [420](#)
CamelDigitalSignature header [611](#)
camel directory [201](#)
camel-disruptor component [570](#)
camel-etcd component [721](#)
CamelExecutionException [612](#), [621](#)
Camel Extra project [654](#), [837](#)
CamelFileName header [38](#), [55](#)
Camel for microservices, with WildFly Swarm [256](#)
camel-ftp component [614](#)
camel-google-calendar component [337](#)
camel-google-drive component [337](#)
camel-google-mail component [337](#)
camel-google-pubsub component [337](#)
camel-hazelcast component [558](#), [721](#), [733](#), [734](#)
camel-http component [614](#)
camel-hystrix-starter dependency [298](#), [790](#)
camelinaction.inventory package [659](#)
camelinaction.routes package [46](#), [47](#)
camelinaction.ServerBar [558](#)
camelinaction.ServerFoo [558](#)
camelinaction package [352](#), [367](#), [659](#)
camel-infinispan module [556](#), [560](#)
camel-irc component [614](#)
camel-jackson, binding JSON to/from POJOs with [419–420](#)
camel-jaxb, binding XML to/from POJOs with [419](#)

CamelJdbcRowCount header [221](#)
CamelJdbcUpdateCount header [221](#)
camel-jetty9 component [614](#)
camel-jetty component [429](#)
camel-jms-starter dependency [298](#)
camel-kafka component [727, 728](#)
camel-linkedin component [337, 614](#)
Camel Log components [687–688](#)
camel-mail component [614](#)
Camel management API [698–712](#)
 accessing using Java [699–703](#)
 using JMX proxy [702–703](#)
 using standard Java JMX API [700–701](#)
 management-enable custom components [708–711](#)
 management-enable custom Java beans [711–712](#)
 performance statistics [705–711](#)
 using from within Camel [703–704](#)
Camel Maven plugin [822](#)
camel-metrics [706–708](#)
camel-mina2 component [614](#)
CamelNamespaceHandler [627](#)
camel-netty4 component [614](#)
camel-netty4-http component [429](#)
camel-netty component [614](#)
camel-olingo2 component [337, 614](#)
camel-olingo4 component [337](#)

camel-package-maven-plugin [322](#)

camel-restlet [417–420](#)

- binding JSON to/from POJOs using camel-jackson [419–420](#)
- binding XML to/from POJOs using camel-jaxb [419](#)

camel-restlet component [417, 420, 429, 615](#)

CamelRouteMBean [705](#)

Camel routes [428](#)

camel-salesforce component [615](#)

CamelServlet [260](#)

camel-servlet component [429](#)

camel-spark-rest component [429](#)

CamelSpringBootRunner [357](#)

CamelSpringDelegatingTestContextLoader class [355](#)

CamelSpringTestSupport class [351, 354](#)

camel-spring-ws component [615](#)

camel-test, testing Java Camel routes with [346–349](#)

camel-test-blueprint module [317](#)

camel-test-karaf module [317](#)

Camel Test Kit [345–349](#)

- JUnit extensions [346](#)
- testing Java Camel routes with camel-test [346–349](#)
- unit testing existing RouteBuilder classes [349](#)

camel-test module [317](#)

camel-test-spring module [317](#)

CamelTestSupport class [346, 347, 350, 370, 642](#)

CamelTimerFiredTime header [231](#)

camel-twilio component [337](#)
camel-undertow component [429](#), [615](#)
CamelVersion [674](#)
camel-version property [14](#)
camel-websocket component [615](#)
camel-zendesk component [337](#)
camel-zookeeper component [721](#)
carbon copy. *See* CC (carbon copy)
CartDto.java file [264](#)
CartDto class [267](#)
CartService class [267](#)
CassandraAggregationRepository [159](#), [163](#)
catching
 exceptions [542–543](#)
 exceptions with onException method [492–495](#)
CBRs (content-based routers) [55–61](#)
 routing after [60–61](#)
 using otherwise clause [58–59](#)
CC (carbon copy) [234](#)
CDI (Contexts and Dependency Injection) [19](#), [247](#), [308](#),
[666–669](#)
 as microservice [247–252](#)
 configuring applications [248–249](#)
 dependency injection using @Inject [249–250](#)
 hello service with [247–248](#)
 listening to events using [251–252](#)

testing with [349–361](#), [362–368](#)
using @Uri to inject endpoints [250–251](#)

CdiBeanRegistry [117](#), [119](#)

CdiCamelContext class [117](#)

ChannelInboundHandler [216](#)

ChannelOutboundHandler [216](#)

choice1 EIP [819](#)

choice block [60](#)

choice method [56](#)

chooseProcessor method [191](#)

cia_keystore.jks file [609](#)

cia_secrets.jceks file [612](#)

CICS (Customer Information Control System) [837](#)

Circuit Breaker EIP

- and Camel [282](#)
- patterns [280–281](#)
- to handle failures [280–282](#)

circuitBreakerErrorThresholdPercentage option [288](#)

circuitBreakerRequestVolumeThreshold option [288](#)

circuitBreakerSleepWindowInMilliseconds option [288](#)

Citrus integration testing, testing applications with [404](#)

Class component [198](#)

class element [225](#)

ClassPathXmlApplicationContext [351](#)

class property [197](#)

client acknowledge mode [521](#)

Client class [607](#)
clients, sending order messages with [391](#)
closeCorrelationKeyOnCompletion option [157](#)
cloud [9](#)
Cloud Foundry [244](#)
Cloud Native Computing Foundation [777](#)
clustered cache [733](#)
clustered idempotent repositories [556–561](#)
 configuring eviction strategies [560–561](#)
 Hazelcast as [558–560](#)
clustered memory grid [733](#)
clustering [717–751](#)
 caches [733–737](#)
 using Hazelcast [733–735](#)
 using Infinispan [736–737](#)
 using JCache [736–737](#)
 calling clustered services using Service Call EIP [739–750](#)
 configuring [745–747](#)
 overview [740–741](#)
 URI templating [747–748](#)
 using Consul [748–750](#)
 using Spring Boot Cloud [748–750](#)
 using static service registry [741–745](#)
 with failover [744–745](#)
 clustered Camel routes [720–726](#)
 active/active mode [720–721](#)

active/passive mode [720](#)
active/passive mode using Consul [723–724](#)
active/passive mode using Hazelcast [721–723](#)
active/passive mode using ZooKeeper [724–726](#)
clustered HTTP [718–719](#)
clustered JMS [726–727](#)
clustered Kafka [727–733](#)
crashing JVM with running Kafka consumers [731–733](#)
Kafka consumer offsets [730–731](#)
clustered scheduling [737–739](#)
counters [735](#)

clusters
Java microservices calling each other in [771–773](#)
Kubernetes [766–768](#)

codecs
custom [216–217](#)
for object serialization [215](#)

CollisionAvoidanceFactor option [483](#)

combineData bean [582](#)

commandKey option [288](#)

command keys, configuring [286](#)

command message [218](#)

commands, calling from Java [283](#)

comma separated values. *See* CSV (comma separated values)

commit method [522](#)

committers, in Camel community [837](#)

commons-net [311](#)

communicating, between Docker containers [760–761](#)

community [10](#), [835–838](#)

 Apache Camel website [835](#)

 at GitHub [836](#)

 at Stack Overflow [836](#)

 Camel committers [837](#)

 Camel Extra project [837](#)

 commercial offerings [836](#)

 in Gitter [835–836](#)

 in IRC [835–836](#)

 in JIRA [835–836](#)

 mailing lists [835–836](#)

 resources [838](#)

 tooling [837](#)

 videos [837](#)

compensating, for unsupported transactions [544–551](#)

 onCompletion [548–551](#)

 synchronization callbacks [546–547](#)

 UnitOfWork [545–546](#)

completion conditions, for Aggregator EIPs [152–159](#)

 aggregated exchange properties [155–156](#)

 configuration options [156](#)

 using multiple completion conditions [154–155](#)

 using POJO for AggregationStrategy [158–159](#)

completionFromBatchConsumer condition [153](#)

completionInterval condition [153](#)

completionPredicate condition [153](#)

completionSize attribute [150](#), [153](#)

completionTimeout condition [153](#)

component

Properties [828](#)

component classes [320–323](#)

Component interface [196](#)

component option [430](#)

componentProperty option [431](#)

component resolver [197](#)

components

ActiveMQ [205](#)

autodiscovering [196–197](#)

example using InOut MEP [209](#)

how BeanComponent is resolved [197](#)

camel-core module [197](#)

configuring [47–48](#)

configuring in Rest DSL [431–434](#)

creating a custom component [318](#)

custom, developing [318–326](#)

Direct components [226–230](#)

Direct-VM components [226–230](#)

File components [198–204](#)

default filename [202](#)

delay option [200](#)

delete option [200](#), [201](#)
exclude option [200](#), [201](#)
fileExist option [200](#)
fileName option [200](#), [202](#)
include option [200](#), [201](#)
locking files [201](#)
maxMessagesPerPoll option [636](#)
move option [200](#), [201](#)
noop option [200](#)
noop property [44](#)
reading and writing files [201](#)
reading files with [200–201–203](#)
recursive option [200](#)
simple expressions for filename [202](#)
writing files with [200–201](#), [203](#)

FTP components [198–204](#)

- accessing remote files with [203–204](#)
- binary option [203](#)
- consuming files [30](#)
- default FTP port [30](#)
- disconnect option [203](#)
- example using embedded FTP server [204](#)
- Maven dependency [30](#), [203](#), [311](#)
- maximumReconnectAttempts option [203](#)
- password [30](#)
- password option [203](#)

reconnectDelay option [203](#)
remote directory [30](#)
transitive dependencies [312](#)
URI options [203](#)
username [30](#)
username option [203](#)
using embedded FTP server for testing [204](#)
generating with API component framework [326–337](#)
 configuring camel-api-component-maven-plugin
[328–330](#)
 generating skeleton API project [326–328](#)
 implementing [335–337](#)
 setting advanced configuration options [331–335](#)
Hibernate [218](#)
implementing [320–326](#)
 component classes [320–323](#)
 consumers [323–326](#)
 endpoint classes [320–323](#)
 producers [323–326](#)
JDBC components [218–226](#)
 accessing data with [218–221](#)
 example [221](#)
 Maven dependency [218, 234](#)
 readSize option [218](#)
 statement option [218](#)
 URI options [218, 234, 235](#)

useHeadersAsParameters option [218](#)
useJDBC4ColumnNameAndLabelSemantics option [218](#)
using a bean to create the SQL statement [220](#)

JMS components [204–212](#)

- ActiveMQConnectionFactory [205](#)
- clientId options [206](#)
- concurrentConsumers options [206](#)
- connecting to ActiveMQ [32](#)

over TCP [204](#)

- disableReplyTo options [206](#)
- durableSubscriptionName options [206](#)
- exchangePattern option [209](#)
- JMSCorrelationID header [209](#)
- JMSReplyTo header [209](#)
- mapJmsMessage option [212](#)
- Maven dependency [32, 206](#)
- maxConcurrentConsumers options [206](#)
- message mappings [210–212](#)
- receiving messages [208–209](#)
- replyTo option [206, 210](#)
- request-reply messaging [209–210](#)
- requestTimeout options [206](#)
- selector options [206](#)
- sending messages [208–209](#)
- sending messages with the ProducerTemplate [181](#)
- transacted options [206](#)

URI options [205](#)
using a topic [208](#)
using multiple providers [47](#)
valid JMS headers [212](#)

components

JPA components [218–226](#)

- connecting Camel JPA to OpenJPA [224](#)
- consumeDelete option [222](#)
- consumeLockEntity option [222](#)
- consumer.delay option [222](#)
- consumer.initialDelay option [222](#)
- consumer.namedQuery option [222](#)
- consumer.nativeQuery option [222](#)
- consumer.query option [222](#)
- example [225](#), [233](#), [236](#)
- flushOnSend option [222](#)
- manually querying the database [226](#)
- maximumResults option [222](#)
- maxMessagesPerPoll option [222](#)
- persistenceUnitName [225](#)
- persistenceUnit option [222](#)
- persisting objects with [221–226](#)
- transactionManager option [222](#)
- URI options [222](#)

LGPL license [654](#)

libraries [6](#)

management-enabled [708–711](#)
manually adding [196](#)
 custom codecs [217](#)
 decoder option [213](#)
 decoders option [213](#)
 encoder option [213](#)
 encoders option [213](#)
 encoding option [213](#)
 example [215](#)
 Maven dependency [213](#)
 setting up a TCP server [213](#)
 sync option [213](#)
 textline codec [215](#)
 textlineDelimiter option [213](#)
 textline option [213](#)
 timeout option [213](#)
 transport types [213](#)
 URI options [213](#)
MyBatis [218](#)
naming in Spring [47](#)
Netty4 components [213–217](#)
 for network programming [213–215](#)
 using custom codecs [216–217](#)
number of components [195](#)
overview [195–197](#)
Properties [53](#)

properties file [53](#)

syntax [51](#)

Quartz2 components [230–233](#)

enterprise scheduling with [231–233](#)

cron option [231](#)

job.name option [231](#)

job.propertyName option [231](#)

JobDetail [231](#)

Maven dependency [231](#)

SimpleTrigger [231](#)

Trigger [231](#)

trigger.propertyName option [231](#)

trigger.repeatCount option [231](#)

trigger.repeatInterval option [231](#)

URI options [231](#)

Scheduler components [230](#), [231–233](#)

SEDA [185](#)

SEDA components [226–230](#)

asynchronous messaging with [228–230](#)

concurrentConsumers option [228](#)

multipleConsumers option [228](#), [229](#)

publish-subscribe [228](#)

size option [228](#)

timeout option [228](#)

URI options [227](#), [228](#)

waitForTaskToComplete option [228](#)

setting up new [318–319](#)

SQL [218](#)

starter, with Spring Boot [259](#)

stream [201](#)

supported in Rest DSL [429](#)

versus WildFly Swarm fragments [254–255](#)

VM components [226–230](#)

working with emails [233–237](#)

XSLT [87](#)

Composed Message Processor EIP [147](#)

computing, routing slip headers with beans [179–180](#)

concurrency [563–586](#)

EIPs [578](#)

- Multicast EIP [581–583](#)
- Threads EIP [578–581](#)
- Wire Tap EIP [583–586](#)

enabling parallelProcessing() [566–567](#)

ExecutorServiceManager [576–578](#)

- configuring Camel to use custom [577](#)
- using in custom component [577–578](#)

running example without [565](#)

SEDA [568–570](#)

concurrency

thread pools

- creating custom [567–568](#), [575–576](#)
- in Java [571–573](#)

profiles 573–575
concurrentConsumers option 569
ConfigMap 783
configurations
 for Aggregator EIPs 156
 importing 48–49
 setting advanced options in Spring 49
configuration security 596–599
 decrypting configuration 597–599
 encrypting configuration 596–597
configure method 34, 35, 37, 57, 347, 349, 383, 564, 746
configuring
 API components 331–335
 global options 331–333
 Javadoc-only options 334–335
 enabling API documentation 459–460
 filtering CamelContext in 460–461
 filtering routes in 460–461
 broker-level redelivery with ActiveMQ 538
 camel-api-component-maven-plugins 328–330
 Camel to use JMS providers 32
 CDI applications 248–249
 command keys 286
 components 47–48
 consumer-level redelivery with ActiveMQ 537
 data formats 97–99

endpoints [47–48](#)
event notifiers [692](#)
eviction strategies [560–561](#)
Hystrix [287–288](#)
 options [288](#)
 with Camel [287–288](#)
 with Java [287](#)
JMX Agent from XML DSL [679–680](#)
ManagementAgent from Java [679](#)
NodePort in pom.xml [770](#)
Quartz to use databases [738](#)
Rest DSL [429–434](#)
 components [431–434](#)
 consumer options [431–434](#)
 endpoints [431–434](#)
 options [430–431](#)
Rest DSL binding [436](#)
Service Call EIP [745–747](#)
Spring Boot with Rest DSL [441](#)
confirm method [553](#)
ConnectException [379, 493](#)
ConnectionException [528](#)
ConnectionFactory instance [31, 32, 196, 204](#)
consoles [816–822](#)
 best practices [821–822](#)
 connecting to existing running JVM [820–821](#)

debugging routes using [818–821](#)
functionality of [817–818](#)

ConstraintSecurityHandler [432](#)

ConsulRoutePolicy [723](#)

Consul tool

- clustered active/passive mode [723–724](#)
- Service Call EIP with [748–750](#)

consumer.bridgeErrorHandler [508](#)

consumerProperty option [431](#)

consumers [23](#), [323–326](#)

- bridging with error handlers [505–509](#)
- event-driven [24](#)

consumers

- Kafka
 - crashing JVM with [731–733](#)
 - offsets [730–731](#)
 - polling [24](#)
- consuming, from FTP endpoints [30–31](#)

containers

- Docker, communicating between [760–761](#)
- exploring while running [761–762](#)
- in Spring [46–49](#)
 - configuring components [47–48](#)
 - configuring endpoints [47–48](#)
 - finding route builders [46–47](#)
 - importing configuration [48–49](#)

importing routes [48–49](#)
setting advanced configuration options [49](#)

contains method [373](#)

Content-Based Router EIP [819](#)

contentBasedRouter method [56](#)

content-based routers. *See* CBRs (content-based routers)

Content Enricher EIPs [151](#)

- enrich method [85, 86](#)
- pollEnrich [85](#)
- transforming data with [84–87](#)

Content-Type header [435, 436](#)

<context:property-placeholder> tag [53](#)

ContextName annotation [667](#)

contextPath option [91, 430](#)

ContextRefreshedEvent event [627](#)

Contexts and Dependency Injection. *See* CDI (Contexts and Dependency Injection)

contextScan element [46](#)

context-scoped error handlers [490–492](#)

continuedProcessing queue [60, 61](#)

Control Bus

- managing applications with [697–698](#)
- starting and stopping routes at runtime [638](#)

controllers. *See* replication controllers

control pane [764](#)

convention over configuration paradigm [8](#)

convertBodyTo method [102](#)
converters [99–105](#)

- adding to Camel-core [104–105](#)
- loading into registry [101–102](#)
- overview [101–102](#)
- using [102–103](#)
- writing [103–105](#)

convertTo method [101](#)
core, lightweight for microservices [9](#)
core logs [685](#)
corePoolSize option [288, 572](#)
<correlationExpression> tag [150](#)
correlation IDs [689](#)
CORS (Cross-Origin Resource Sharing) [461–465](#)
counters, clustered [735](#)
CPU cores [761](#)
createApplicationContext method [353](#)
createConnector option [700](#)
create method [394](#)
createOrder operation [422, 447](#)
createSecurityHandler method [433](#)
cron expressions [233, 737, 738](#)
cron option [233](#)
cron triggers [233](#)
cryptographic protocols [203](#)
CSV (comma separated values)

data formats, using [92–94](#)
table of data types used [94](#)
transforming using Processor [79](#)

csvOrders queue [57](#)

curl command [673](#)

custom component [318](#)
data marshaling [324](#)
disabling producer or consumer creating [323](#)
extending from default implementations [323](#)

Customer Information Control System. *See* CICS (Customer Information Control System)

CustomerService.class [140](#)

CustomerService bean [171](#)

custom logging [685](#)

Custom strategy [186](#)

CXF. *See* Apache CXF

CxfConverter [102](#)

cxfEndpoint bean [601](#)

CxfPayloadConverter [102](#)

CXFServlet [652](#)

D

data
accessing with JDBC components [218–221](#)
enriching with pollEnrich [85–87](#)
transforming [75–105](#)

overview [76–77](#)

transforming XML [87–91](#)

- using <transform> in XML DSL [83](#)
- using beans [80–81](#)
- using EIPs [77–87](#)
- using Java [77–87](#)
- using processors [78–80](#)
- using transform method in Java DSL [81–](#)
- with data formats [91–99](#)
- with templates [99](#)
- with type converters [99–105](#)

databases

- configuring to use Quartz [738](#)
- resources, transactions starting from [534–536](#)
- setting up [518–519](#)
- setting up for Quartz [738](#)
- transaction redeliveries starting from [538–539](#)

dataFormatProperty option [436](#)

data formats

- Bindy, using [94–97](#)
- configuring [97–99](#)
 - configure [97](#)
 - consume files [92](#)
- custom, writing [339–342](#)
- DataFormat API [339](#)
- generating projects [338–339](#)

JSON [97](#), [434–443](#)
marshal [89](#)
providing [92](#)
transforming with [91–99](#)
unmarshal [89](#)
using JAXB [91](#)
using XStream [89](#)
writing custom, using Camel type converters [340](#)

XML, with Rest DSL [434–443](#)

data format transformation [76](#)

data integration, and data mapping [77](#)

data mapping [77](#)

DataSet component [198](#)

DataSourceTransactionManager [536](#)

data type transformation [76](#), [77](#)

DeadLetterChannel error handler

- enriching messages with cause of error [478–480](#)
- handling exceptions by default [477](#)
- useOriginalMessage with [478](#)

dead letter queue [475](#), [476–477](#), [478](#), [479](#), [480](#), [489](#), [501](#), [513](#)

deadLetterUri option [166](#)

debuggers, integrated [806–808](#)

debugging

- Java applications in Kubernetes [774–776](#)
 - calling service in Kubernetes changing port numbers [775–776](#)

debugging running pods [774–775](#)
projects [316–318](#)
routes using hawtio [818–821](#)
unit tests [317–318](#)

DEBUG logging level [685](#)

declarative states [778](#)

DefaultAsyncProducer [592](#)

DefaultCamelContext class [42, 115](#)

DefaultComponent [320](#)

DefaultConsumer class [325](#)

DefaultEndpoint class [323](#)

DefaultErrorHandler

- overview [474–475](#)
- redelivery with [483–487](#)

DefaultExecutorServiceManager class [576](#)

DefaultManagementAgent class [679](#)

DefaultProducer [324](#)

- defaultProfile attribute [574](#)
- defaultValue option, JSON schema [816](#)
- defining routes in XML [39–49](#)
 - creating applications from beans using Spring [40–43](#)
 - running Camel in Spring containers [46–49](#)
 - XML DSLs [43–45](#)

Delayer EIP [578](#)

Delayer option [629](#)

delay option [230, 231](#)

delete option [201](#)

DELETE verb [410](#)

deleting pods [784](#)

dependencies

 adding with Maven [311–313](#)

 injecting with `@Inject` [249–250](#)

dependency:tree command [312](#)

deploying Camel [645–658](#)

 embedded in Java application [645–648](#)

 embedded in web application [648–654](#)

 deploying to Apache Tomcat [650–652](#)

 using Apache Tomcat for HTTP inbound endpoints
[652–654](#)

 embedded in WildFly [654–658](#)

 deploying Camel

 in a Java application

 file copy example [646](#)

 pros and cons [647](#)

 in a web application [648](#)

 pros and cons [653](#)

 using Spring context listener [648](#)

 in OSGi, pros and cons [665](#)

 in WildFly, pros and cons [655](#)

 in WildFly-Camel, pros and cons [658](#)

deployment strategy

 Apache Tomcat [653](#)

Java application [647](#)

Jetty [650](#)

OSGi [665](#)

standalone [645](#)

WildFly [655](#)

WildFly-Camel [658](#)

deployment units, writing test classes with [398–399](#)

deprecated option, JSON schema [816](#)

DES (Digital Encryption Standard) [613](#)

describe command [780, 784](#)

description option, JSON schema [816](#)

designing microservices for failures [277–](#)

- bulkhead patterns [289–292](#)
- calling microservices with fault-tolerance [292–298](#)
- using Circuit Breaker [280–282](#)
- using Hystrix [282–288, 301–305](#)
- using Hystrix with Spring Boot [298–301](#)
- using Retry patterns [277–279–280](#)

destinations, specifying with URIs [33](#)

digital signatures

- generating and loading public and private keys [609–610](#)
- signing and verifying messages using camel-crypto [610–612](#)

direct:dead route [480](#)

direct:start endpoint [576](#)

direct:startOrder endpoint [227](#)

Direct components [226–230](#)

Direct-VM components [226–230](#)

displayName option, JSON schema [816](#)

distributed tracing [691](#)

distribution, obtaining [10](#)

Dname=HiWorld argument [330](#)

dname argument [609](#)

doCatch [498](#)

doc directory [10](#)

Docker CLI, with Kubernetes [764](#)

docker images command [757](#)

Docker Maven

building Docker images using plugins [758–759](#)

fabric8 plugin [756–757](#)

running Java applications in Kubernetes with [768–770](#)

Docker platform

building images using Docker Maven plugins [758–759](#)

communicating between containers [760–761](#)

microservices with [752–753, 762–801](#)

running Java microservices on [759–762](#)

communicating between Docker containers [760–761](#)

exploring running containers [761–762](#)

Docker Swarm [244](#)

documenting

APIs, using Swagger for [444–465](#)

body output types [455–456](#)

error codes [454–457, 459](#)

header output types 457
input 454–459
output 454–459
RESTful web services 452–453
doFinally 498
domain-specific language. *See* Rest DSL (domain-specific language)
domain-specific languages. *See* DSLs (domain-specific languages)
done method 593
doneSync parameter 590
doStart method 577, 592, 693
doStop method 577, 693
doTry 498
double binding 435
doubleUp method 125
Dropwizard metrics 671, 706, 707
DSLs (domain-specific languages)
adding processors 38–39
choosing 45
in Java 81–83
XML 43–45
adding processors 44–45
invoking beans defined in 108–109
using <transform> to transform data 83
using multiple routes 45
using property placeholders in 51–52

with Spring Boot [260–261](#)
dumb pipes [244](#)
Dumpleton, Graham [801](#)
dumpStatisticsAsJson operation [707](#)
dumpStatsAsXml(boolean) method [706](#)
dynamic endpoints, sending to [49–50](#)
dynamic parts [50](#)
Dynamic Router EIPs
 overview [148, 182](#)
 using @DynamicRouter annotation [182–184](#)
dynamic to, using toD as [143–145](#)

E

eagerCheckCompletion option [157](#)
EAI (Enterprise Application Integration) [244](#)
EchoBean class [120](#)
echo method [120, 123, 124, 125, 128, 130](#)
Eclipse Equinox [658](#)
Eclipse IDE
 overview [306](#)
 using Camel in [313–315](#)
EclipseLink [221](#)
editors [804–813](#)
 Apache Camel IDEA plugins [808–810](#)
 Camel validation using Maven [811–813](#)
 graphical [804–805](#)

JBoss Fuse Tooling [804–808](#)
type-safe endpoint [806](#)

EIPs (enterprise integration patterns)

- Aggregator EIPs [148–149](#), [152–166](#)
 - completion conditions for [152–159](#)
 - overview [147](#)
 - using AggregationStrategy [151–152](#)
 - using persistence with [159–163](#)
 - using recovery with [163–166](#)
- command message [218](#)
- Composed Message Processor [147](#), [174](#)
- content-based router [56](#)
- Content Enricher EIPs [84–87](#), [151](#)
- Control Bus, starting and stopping routes at runtime [638](#)
- Dead Letter Channel [165](#)
- Dynamic Router EIPs [182–184](#)
 - overview [148](#), [182](#)
 - using @DynamicRouter annotation [182–184](#)
- implemented as Processor [38](#)
 - overview [148](#), [184–186](#)
 - using custom load balancers [190–193](#)
 - using failover load balancer [188–190](#)
 - using load-balancing strategies [186–187](#)
- Message Translator EIP [78–83](#)
- multicast [63](#)
 - configuring the thread pool [64](#)

implement with a JMS topic [208](#)

parallelProcessing [64](#)

overview [146–148](#)

parallel processing [578](#)

- Multicast EIP [581–583](#)
- Threads EIP [578–581](#)
- Wire Tap EIP [583–586](#)

comma-separated recipients [66](#)

example [68](#)

routing and [55–71](#)

- using CBR [55–61](#)
- using message filters [61–62](#)
- using multicasting [62–65](#)
- using recipient lists [66–69](#)
- using wireTap method [69–71](#)

Routing Slip EIPs [178–182](#)

overview [148, 179](#)

using @RoutingSlip annotations [180–182](#)

using beans to compute routing slip headers [179–180](#)

using Expressions as routing slips [180](#)

Splitter EIPs [166, 170](#)

aggregating split messages [174–175](#)

errors during splitting [176–178](#)

overview [147, 169–170](#)

splitting messages [172–173](#)

using beans for splitting [170–172](#)

transforming data with [77–87](#)

EL language [56](#)

emails [233–237](#)

- receiving with IMAP [235–237](#)
- sending with SMTP [234–235](#)

embedded FTP server [204](#)

embedded HTTP server [755](#)

embedding

- in Java application [645–648](#)
- in web application [648–654](#)
- in WildFly [654–658](#)

Swagger UI web console [463–465](#)

emptyCart method [293](#)

enableCORS option [459](#)

enabling API documentation [459–460](#)

ENC() notation [597](#)

encryption

- decrypting configuration [597–599](#)
- encrypting configuration [596–597](#)
- payload encryption [612–614](#)
 - encrypting and decrypting payloads using camel-crypto [613–614](#)
 - generating and loading secret keys [612–613](#)

endpoint classes [320–323](#)

endpoint editor [821](#)

EndpointMessageListener [526](#)

endpointProperty option [431](#)

automocking [385](#)

Camel-configured [420–422](#)

configuring [47–48](#)

configuring in Rest DSL [431–434](#)

consuming from FTP endpoints [30–31](#)

dynamic, sending to [49–50](#)

mock [168](#)

replacing routes [385–387](#)

SEDA [186](#)

sending to JMS endpoints [31–33](#)

configuring Camel to use JMS providers [32](#)

using URIs to specify destinations [33](#)

Spring Boot REST [258](#)

endpoints

URIs

referencing registry beans in [54](#)

using property placeholders in [50–53](#)

using raw values in [54](#)

<endpoint> tags [53](#)

Endpoint URI [22, 30](#)

endsWith method [58, 373](#)

enriching

data with pollEnrich [85–87](#)

messages with cause of error [478–480](#)

Enterprise Application Integration. *See* EAI (Enterprise

Application Integration)

enterprise integration patterns. *See* EIPs (enterprise integration patterns)

enterprise resource planning. *See* ERP (enterprise resource planning)

enterprise scheduling, with Quartz2 components [231–233](#)

enterprise service bus. *See* ESB (enterprise service bus)

entity manager [224](#)

EntityManagerFactory [226](#)

ERP (enterprise resource planning) [542](#)

ERPEndpoint class [708](#)

ErpProducer class [591, 593](#)

ERP system [563, 564, 565, 566, 586, 587, 588, 589, 591, 593](#)

ERPTask [592](#)

error codes, documenting [454–457, 459](#)

errorHandlerRef attribute [489, 490](#)

error handlers [473–481](#)

 bridging consumers with [505–509](#)

 context-scoped [490–492](#)

 DeadLetterChannel [475–476–477, 480](#)

 enriching messages with cause of error [478–480](#)

 handling exceptions by default [477](#)

 useOriginalMessage with [478](#)

 DefaultErrorHandler [474–475](#)

 features of [481](#)

 HandleFault [631](#)

 implementing [504–505](#)

Load Balancer EIP failover [189](#)

LoggingErrorHandler [481](#)

NoErrorHandler [480](#)

error handlers

 redelivery with

 using DefaultErrorHandler with [483–487](#)

 using redelivery [482–483](#)

 scopes and [487–490](#)

 TransactionErrorHandler [480, 530](#)

<errorHandler> tag [490](#)

errors

 cause of, enriching messages with [478–480](#)

errors. *SeeAlso* error handlers

 exception policies [492–509](#)

 overview [470–473](#)

 using onExceptionOccurred [510–511](#)

 using onRedeliver [511](#)

 using onWhen [509–510](#)

 using retryWhile [512](#)

 where applicable [473](#)

errors

 irrecoverable [470–472](#)

 recoverable [470–472](#)

 simulating [378–390](#)

 adding interceptors to existing routes with adviceWith method [382–383](#)

amending routes using weave with adviceWith method
[387–390](#)

manipulating routes with adviceWith method [383–387](#)
with interceptors [381–382](#)
with mocks [380–381](#)
with processors [378–380](#)

ESB (enterprise service bus) [5, 244](#)

event-driven consumers [24](#)

Event message [17](#)

event notifiers [692–694](#)

EventNotifierSupport class [693](#)

events, listening with CDI [251–252](#)

eviction, configuring strategies [560–561](#)

examples directory [10](#)

ExceptionHandler [640](#)

Exception parameter [130](#)

exception policies [492–509](#)

 bridging consumers with error handlers [505–509](#)

 custom exception handling [500–502](#)

 ignoring exceptions [503–504](#)

 implementing error handler solutions [504–505](#)

 new exceptions while handling exception [502–503](#)

exception policies

 onException

 catching exceptions with [492–495](#)

 handling exceptions with [497–498–500](#)

multiple exceptions per [495](#)
redelivery and [496–497](#)

exceptions

- catching [492–495, 542–543](#)
- handling [500–502, 542–543](#)
- handling by default [477](#)
- handling using AggregationStrategy [176–178](#)
- handling with Rest DSL [441–443](#)
- ignoring [503–504](#)
- multiple per onException [495](#)
- new exceptions while handling [502–503](#)
- rolling back [543](#)
- stopping multicast on [64–65](#)

exception tag [500](#)

exchange

- exchange ID [17](#)
- for messages [16–18](#)
- in message [17](#)
- instance, working on via Processor [78](#)
- MEP [17](#)
- out message [17](#)
- properties [17](#)

Exchange.AGGREGATED_COMPLETED_BY property [156](#)

Exchange.AGGREGATED_CORRELATION_KEY property [156](#)

Exchange.AGGREGATED_SIZE property [156](#)

Exchange.AGGREGATED_TIMEOUT property [156](#)
Exchange.AGGREGATION_COMPLETE_ALL_GROUPS_INCLUSIVe property [156](#)
Exchange.AGGREGATION_COMPLETE_ALL_GROUPS property [156](#)
Exchange.AGGREGATION_COMPLETE_CURRENT_GROUP property [156](#)
Exchange.ERRORHANDLER_HANDLED property [501](#)
Exchange.EXCEPTION_CAUGHT property [477, 500, 501](#)
Exchange.FAILURE_ENDPOINT property [501](#)
Exchange.FAILURE_HANDLED property [501](#)
Exchange.FILE_NAME_PRODUCED header [546, 549](#)
exchange.getException() method [500](#)
Exchange.REDELIVERED header [166, 487](#)
Exchange.REDELIVERY_COUNTER header [166, 487](#)
Exchange.REDELIVERY_DELAY header [166](#)
Exchange.REDELIVERY_EXHAUSTED header [166, 487](#)
Exchange.REDELIVERY_MAX_COUNTER header [166](#)
Exchange.SPLIT_COMPLETE property [170](#)
Exchange.SPLIT_INDEX property [170](#)
Exchange.SPLIT_SIZE property [170](#)
ExchangeId() method [689](#)
Exchange parameter [130](#)
exchange properties [155–156](#)
ExchangesTotal [674](#)
excludeClasses [334](#)
excludeConfigNames [333](#)

excludeConfigTypes [333](#)
excludeMethods [334](#)
excludePackages [334](#)
Exec probe [786](#)
executionTimeoutInMilliseconds option [288](#)
ExecutorService interface [571](#)
ExecutorServiceManager interface [576–578](#)
 configuring Camel to use custom [577](#)
 using in custom component [577–578](#)
executorService option [578](#)
executorServiceRef attribute [567, 576, 583](#)
exhausted redelivery attempts [486](#)
exit command [762](#)
expectedBodiesReceivedInAnyOrder method [370, 372](#)
expectedBodiesReceived method [370, 371](#)
expectedMessageCount method [370, 371](#)
expectedMinimumMessageCount method [370](#)
expectsAscending method [373](#)
expectsDescending method [373](#)
expectsDuplicates method [373](#)
expects method [373](#)
expectsNoDuplicates method [373](#)
-exportcert keytool command [609](#)
expression builder methods [56](#)
expression language [823](#)
expressions

compound [82](#)
custom [82](#)
in routes, using beans as [138–141](#), [143–145](#)–
method call [83](#), [171](#)
using as routing slips [180](#)
using scripting language as [83](#)
using with mock components [372–376](#)
extensibility [247](#)
extensible markup language. *See* Spring XML (extensible
markup language)
extensible markup language. *See* XML (extensible markup
language)
external DSL [35](#)
extraOptions [333](#)

F

fabric8:build command [765](#)
fabric8-maven-plugin [768](#), [769](#), [770](#), [772](#), [775](#), [776](#), [787](#),
[796–797](#), [798](#), [800](#)
fabric8 project [811](#)
failIfNoConsumers option [227](#)
failover
 Load Balancer EIP [186–190](#)
 Service Call EIP with [744–745](#), [749](#)
 priority based [190](#)
 round robin [189](#)
Failover strategy [186](#)

`FAILURE_ENDPOINT` property [501](#)

failures

dealing with by calling services in Kubernetes [789–792](#)

designing microservices for [277–305](#)

bulkhead patterns [289–292](#)

calling microservices with fault-tolerance [292–298](#)

using Hystrix [282–284](#), [287–288](#), [301–305](#)

using Hystrix with Camel [284–286](#)

using Hystrix with Spring Boot [298–301](#)

using Circuit Breaker to handle [280–282](#)

Circuit Breaker and Camel [282](#)

Circuit Breaker patterns [280–281](#)

using Retry patterns to handle [277–279](#)

how often to retry [278](#)

idempotency [278–279](#)

monitoring [279](#)

SLAs [279](#)

time-outs [279](#)

when to use [278](#)

without Camel [279–280](#)

fallback

adding [283–284](#), [285–286](#)

via networks [292](#)

with Hystrix [290–291](#)

fallback type converter [419](#)

`FataFallbackErrorHandler` [502](#)

[fault flag](#) [16](#)

[fault messages](#) [16](#)

[fault-tolerance, calling microservices with](#) [292–298](#)

examples of [295–296](#), [297–298](#)

using Hystrix with Camel [296–297](#)

using Hystrix with Spring Boot [293–295](#)

using Hystrix with WildFly Swarm [296–297](#)

[FileComponent header](#) [22](#), [38](#)

[File components](#) [198–204](#)

overview [198](#), [545](#)

reading files with [200–201–203](#)

writing files with [200–203](#)

[FileCopier](#) [12](#)

[FileCopierWithCamel class](#) [14](#)

[FileEndpoints](#) [22](#)

[FileIdempotentRepository](#) [555](#)

[FileInventoryServiceFactory](#) [663](#)

[FileMoveRoute class](#) [349](#), [352](#), [353](#)

[fileName option](#) [202](#)

[FileNotFoundException](#) [495](#)

[FileProducer](#) [23](#)

[FileRollback class](#) [547](#), [548](#)

files

autogenerating [767](#)

reading with File components [200–201–203](#)

remote, accessing with FTP components [203–204](#)

variables, in Simple [828–830](#)
writing with File components [200–201](#), [203](#)

filesystems [198](#)

FileSystemXmlApplicationContext [351](#)

file transfer [198](#)

file transfer protocol. *See* FTP (file transfer protocol)

filtering API documentation [460–461](#)

filter method [394](#)

findByType method [113](#)

findByNameWithMethod method [113](#)

FirstTest class [347](#), [362](#)

FirstWildFlyIT class [366](#)

flexibility [645](#)

flipPolicy bean [641](#)

fluent builders [36](#)

fluent interface [36](#)

FluentProducerTemplate [250](#), [258](#), [272](#), [299](#), [345](#)

FooRoute class [357](#)

forceCompletionOnStop condition [153](#)

Fowler, Martin [36](#)

fragments [254](#)

from method [35–36](#), [37](#), [44](#), [394](#)

fromRoute method [394](#)

FTP (file transfer protocol) [30–31](#)

FTP components [198–204](#)

accessing remote files with [203–204](#)

overview [30](#), [38](#)
FTPEndpoint [30](#)
FTP server [28](#)
FtpToJMSRoute class [43](#), [46](#)
functions, in Simple [825–828](#)

G

Gamma, Erich [78](#)
Gatling testing framework [405](#)
GeneralErrorProcessor [502](#)
getAddress method [832](#), [833](#)
getBlueprintDescriptor method [360](#)
getBody method [99](#)
getCamelUptime method [703](#)
getCamelVersion method [703](#)
getContext method [78](#)
getException method [177](#), [471](#)
getExchanges method [392](#)
getFallback method [283](#)
getForObject method [265](#)
getIn method [80](#)
getManagedProcessor method [704](#)
getMockEndpoint method [370](#)
getOut method [80](#)
getReceivedExchanges method [373](#)
getStatusCode method [510](#)

getType method [833](#)
getUnitOfWork method [545](#)
GET verb [410](#)
Ghosh, Debasish [36](#)
GitHub website, Camel community at [836](#)
Gitter messaging, for Camel community [835–836](#)
global transactions [531–534](#), [560](#)
Goetz, Brian [571](#)
Google [777](#)
GoogleMail component, email [234](#)
graphical editors [804–805](#)
graphical tooling [821](#)
Greeting attribute [712](#)
greetMe method [331](#), [334](#)
Groovy [56](#)
groupId=mygroup option [729](#)
groupKey option [288](#)
group option, JSON schema [816](#)

H

Hall, Richard S. [39](#), [658](#)
HandleFault option [629](#), [631](#)
handleIncomingOrder method [134](#)
handle method [602](#)
handlers option [433](#)
handleXML method [126](#), [127](#)

handling. *SeeAlso* error handlers

exception policies [492–509](#)

overview [470–473](#)

using `onExceptionOccurred` [510–511](#)

using `onRedeliver` [511](#)

using `onWhen` [509–510](#)

using `retryWhile` [512](#)

where applicable [473](#)

handling

exceptions [500–502, 542–543](#)

by default [477](#)

new exceptions while [502–503](#)

with `onException` method [497–498–500](#)

happy page [719](#)

hardcoding externalize dynamic parts [50](#)

HATEOAS (Hypermedia as the Engine of Application State)
[452](#)

Hawkular [672, 691](#)

HawtDB, transaction [165](#)

hawtio web console [816–822](#)

best practices [821–822](#)

connecting to existing running JVM [820–821](#)

debugging routes using [818–821](#)

functionality of [817–818](#)

managing application lifecycles with [695–697](#)

hawto:run plugin [820](#)

HazelcastAggregationRepository [159](#), [163](#)
Hazelcast grids
 as clustered idempotent repository [558–560](#)
 clustered active/passive mode [721–723](#)
 clustered cache using [733–735](#)
HazelcastIdempotentRepository [556](#)
header method [56](#)
headers [38](#)
 in messages [16](#)
 in routing slips [179–180](#)
 mapping [212](#)
health
 checking at application levels [675–676](#)
 checking at JVM levels [675](#)
 checking at network levels [672–675](#)
health checks [672](#), [745](#), [755](#), [787](#)
heapster add-on, Minikube [773](#)
HelloBean class [249](#)
HelloCamel class [246](#)
HelloConfiguration class [248](#)
hello method [107](#), [110](#), [258](#)
HelloRestController class [258](#)
HelloRoute class [250](#)
hello service, with CDI [247–248](#)
HelloServlet class [365](#)
hello world [11](#)

Helm, Richard [78](#)

Helm tool

- installing [797](#)
- installing applications using [797–798](#)
- platforms built on top of Kubernetes [798](#)

Hibernate [221, 517](#)

HiWorldFileHello class [329](#)

Hohpe, Gregor [7, 529](#)

hostname option [430](#)

HP OpenView [672](#)

htmlBean class [83](#)

HTTP (hypertext transfer protocol) [718–719](#)

HTTP Get probe [786](#)

HTTP inbound endpoints, using Apache Tomcat for [652–654](#)

HttpOperationFailedException [510](#)

HTTP server, embedded [755](#)

HTTP status codes [441–443](#)

Huss, Roland [801](#)

Hypermedia as the Engine of Application State. *See* HATEOAS
(Hypermedia as the Engine of Application State)

- HystrixCommand class [282, 285](#)
- Hystrix library, using as circuit breaker [790–791](#)
- Hystrix streams [301](#)

/

IBM Tivoli [672](#)

Ibryam, Bilgin [801](#)

id bean [47](#)

IDEA plugins [808–810](#)

idempotency

Idempotent Consumer EIP [552–555](#)

idempotent repositories [555–556](#)

clustered [556–561](#)

FileIdempotentRepository [555](#)

HazelcastIdempotentRepository [556](#)

InfinispanIdempotentRepository [556](#)

JdbcMessageIdRepository [555](#)

JpaMessageIdRepository [556](#)

MemoryIdempotentRepository [555](#)

idempotent consumer [561](#)

Idempotent Consumer EIPs [552–555](#)

IdempotentRepository [552](#)

id parameter [97](#)

ID properties [51](#)

IDs (identifiers), weaving without [388–389](#)

IfReplyExpected value [228](#)

ignoreInvalidCorrelationKeys option [157](#)

ignoring exceptions [503–504](#)

ILLEGAL DATA [509, 510](#)

images

building from sources [798–799](#)

Docker, building using Docker Maven plugins [758–759](#)

IMAP (Internet Message Access Protocol) [235–237](#)

implementing

components [320–326](#)

component classes [320–323](#)

consumers [323–326](#)

endpoint classes [320–323](#)

producers [323–326](#)

error handlers [504–505](#)

-importcert keytool command [609](#)

importing

configurations [48–49](#)

routes [48–49](#)

inBody URI option [330](#)

includeMethods [334](#)

includeStaticMethods [335](#)

incoming message [435, 436](#)

incomingOrders queue [29, 33, 45, 55, 601](#)

increment operation [735](#)

IndexOutOfBoundsException [833](#)

InfinispanIdempotentRepository [556](#)

Infinispan platform, clustered cache using [736–737](#)

in-flight messages [641, 642, 645](#)

in-flight registry [636](#)

INFO logging level [685, 687](#)

inheritErrorHandler option [189](#)

initialDelay option [230](#)

injecting dependencies with `@Inject` [249–250](#)

in-memory messaging [195](#)

in method [373](#)

InOnly MEP [21, 215](#)

InOut MEP [21, 209, 228](#)

input, documenting

- input body parameter types [454–455](#)
- input parameter types [455](#)
- input path parameter types [454–](#)

InputStream [102](#)

inspecting

- messages [69](#)
- queues [392–393](#)

installing

- applications using Helm [797–798](#)
- Camel [10–11](#)
- Helm [797](#)
- Minikube [762](#)
- WildFly [367–368](#)

Integer.MAX_VALUE [228](#)

integration framework [5](#)

integration platform as a service. *See* iPaaS (integration platform as a service)

integration testing [390–395](#)

- inspecting queues [392–393](#)
- NotifyBuilder and [393–395](#)

sending order messages using clients [391](#)
waiting for applications to process messages [392](#)
writing classes [402–403](#)

interceptFromEndpoint [382](#)

interceptors
 adding to routes with adviceWith method [382–383](#)
 simulating errors with [381–382](#)

interceptSendToEndpoint [382](#)

internal DSLs [35](#)

internal services [628](#)

Internet relay chat. *See* IRC (Internet relay chat)

InvalidOrderIdException [472](#)

inventory back end [270–271](#)

InventoryMain class [647](#)

inventory prototype [270–273](#)
 inventory back end [270–271](#)
 running inventory services [273](#)

InventoryService class [564, 668](#)

inventory services [273](#)

Inversion of Control. *See* IoC (Inversion of Control)

invoking beans
 defined in XML DSL [108–109](#)
 from Java [108](#)

IoC (Inversion of Control) [39](#)

IOConverter class [102](#)

IOException [188](#)

IoT (internet-of-things) 5
IRC (Internet relay chat) 835–836
irrecoverable errors 470–472
isEnabled method 693
isEqualTo method 373
isGold method 140
isGreaterThan method 373
isGreaterThanOrEqual method 373
isIllegalData method 510
isInstanceOf method 373
isLessThan method 373
isLessThanOrEqual method 373
isNotEqualTo method 373
isNotNull method 373
isNull method 373
isSilver method 140
isUseAdviceWith method 385, 386
IT, naming in 387
ItemDto.java file 264
its TransactionManager API 522

J

JAAS (Java Authentication and Authorization Service)
604–607
JAASUsernameTokenValidator 604, 605
Jaeger (Uber) 691

JAVA_ENABLE_DEBUG variable [774](#)

java.io.File [127](#)

java.io.FileNotFoundException [494](#), [495](#)

java.io.InputStream type [584](#)

java.io.IOException [470](#), [494](#), [495](#), [511](#), [512](#)

java.lang.AssertionError [370](#)

java.lang.Object [16](#), [139](#)

java.lang.String [16](#), [76](#), [127](#)

java.lang package [334](#)

java.net.ConnectException [493](#)

java.security.SignatureException [611](#)

java.util.concurrent package [571](#)

java.util.Date [828](#)

java.util.Executors factory [567](#)

-javaagent argument [684](#)

Java API for RESTful Web Services. *See* JAX-RS (Java API for RESTful Web Services)

Java applications, embedding Camel in [645](#)–[648](#)

Java beans, management-enabled [711](#)–[712](#)

Java Business Integration. *See* JBI (Java Business Integration)

Java Camel routes [346](#)–[349](#)

Java Cryptography Architecture. *See* JCA (Java Cryptography Architecture)

Java Database Connectivity. *See* JDBC (Java Database Connectivity) components

Java Development Kit. *See* JDK (Java Development Kit)

Javadoc, configuration options [334](#)–[335](#)

Java DSL 34, 575

Java Management Extensions. *See JMX (Java Management Extensions)*

Java Message Service. *See JMS (Java Message Service)*

Java Mission Control. *See JMC (Java Mission Control)*

Java Naming and Directory Interface. *See JNDI (Java Naming and Directory Interface) registry*

Java Persistence API. *See JPA (Java Persistence API) components*

Java programming language

accessing Camel management API 699–703

 using JMX proxy 702–703

 using standard Java JMX API 700–701

 calling commands from 283

 configuring Hystrix with 287

 configuring ManagementAgent from 679

 creating routes in 33–39

 using DSLs 35–39

 using RouteBuilder 34–35

 debugging applications in Kubernetes 774–776

 calling service in Kubernetes changing port numbers
775–776

 debugging running pods 774–775

 DSL 81–83

 invoking beans from 108

 microservices calling each other in clusters 771–773

 running applications in Kubernetes using Maven tooling
768–770

running microservices on Docker [759–762](#)
communicating between Docker containers [760–761](#)
exploring running containers [761–762](#)
transforming data with [77–87](#)
 using Content Enricher EIP [84–87](#)
 using Message Translator EIP [78–83](#)
 using compound predicates in [141](#)
 using Simple from custom code [833](#)
 with Hystrix [282](#)

JavaScript [56](#)
 JavaScript Object Notation. *See* JSON (JavaScript Object Notation)

 Java Secure Socket Extension. *See* JSSE (Java Secure Socket Extension)

 Java thread pools [571–573](#)

 Java Transaction API. *See* JTA (Java Transaction API)

 javaType option, JSON schema [816](#)

 Java Virtual Machine. *See* JVM (Java Virtual Machine)

 javax.inject.Named annotation [668](#)

 javax.jms.Message [23, 211](#)

 javax.jms.MessageListener bean [122](#)

 javax.jms.TextMessage [76](#)

 javax.persistence.Entity [222](#)

 javax.sql.DataSource [218](#)

 javax.ws.rs.core.Application [449](#)

 jaxb.index file [91](#)

JAXB (Java Architecture for XML Binding)

 @XmlAccessorType [90](#)

 @XmlAttribute annotation [90](#)

 @XmlRootElement annotation [90](#)

 contextPath [91](#)

 using annotations [90](#)

JAX-RS (Java API for RESTful Web Services)

 using Camel in existing applications [414–417](#)

 with RESTful web services [410–414](#)

 with Swagger [446–451](#)

JAX-RS Response type [422](#)

JAXRSServerFactoryBean [416](#)

JBIG (Java Business Integration) [472](#)

JBOSS_HOME variable [367](#)

JBoss component [654, 655](#)

JBoss Fuse [519, 529, 534, 658](#)

JBoss Fuse Tooling [804–808](#)

 graphical editors [804–805](#)

 integrated tracers and debuggers [806–808](#)

 type-safe endpoint editors [806](#)

JBoss Tools [317](#)

JBoss Weld project [117, 247](#)

JCA (Java Cryptography Architecture) [596, 613](#)

JCache [556](#)

JCache API, clustered cache using [736–737](#)

JCacheComponent [736](#)

JCachingProvider [736](#)

JConsole tool

- managing Camel with [676–678](#)
- remotely managing Camel with [678–680](#)
- configuring JMX Agent from XML DSL [679–680](#)
- configuring ManagementAgent from Java [679](#)
- using JVM properties [679](#)

JDBC (Java Database Connectivity) components [218–226](#)

JdbcAggregationRepository [159, 163](#)

JdbcMessageIdRepository [555](#)

JDK (Java Development Kit) [609](#)

Jersey [414](#)

Jetty

- deploying Camel in [650](#)
- overview [97](#)

JettySecurity class [432, 433](#)

JIRA project management [835–836](#)

JMC (Java Mission Control) [405](#)

JMH toolkit, testing applications with [405](#)

jms:queue:dead [478](#)

jms:topic:quote endpoint [371](#)

jms.redeliveryPolicy [537](#)

JMS (Java Message Service)

- BytesMessage [211](#)
- clustered [726–727](#)
- JMS message implementations [210](#)

MapMessage [211](#)
ObjectMessage [211](#)
overview [31](#)
providers, configuring Camel to use [32](#)
replacing with stubs [371](#)
sending to endpoints [31–33](#)
setting up brokers [518–519](#)
TextMessage [211](#)
JmsComponent.jmsComponent-AutoAcknowledge method
[196](#)
JMS components [204–212](#)
message mappings [210–212](#)
 body mapping [210–212](#)
 header mapping [212](#)
receiving messages [208–209](#)
request-reply messaging [209–210](#)
 sending messages [208–209](#)
jmsConnectionFactory [524](#)
JMS destination [31](#)
JMSEException [498](#)
JmsProducer [23](#)
JMSRedelivered header [528](#)
JMSReplyTo header [206, 209, 210](#)
JmsTransactionManager [523, 524, 530](#)
JMX (Java Management Extensions) [672, 676–684](#)
 managing Camel remotely with JConsole [678–680](#)

configuring JMX Agent from XML DSL [679–680](#)
configuring ManagementAgent from Java [679](#)
using JVM properties [679](#)
managing Camel with JConsole [676–678](#)
managing Camel with Jolokia [680–684](#)
embedding Jolokia WAR Agent into existing WAR
applications [681–682](#)
 using Jolokia Java agents [683–684](#)
 using Jolokia WAR agents [681–684](#)
 using Jolokia with Apache Karaf [682](#)
using proxy [702–703](#)
using standard API [700–701](#)
JMX Agent, configuring from XML DSL [679–680](#)
JMX attributes [705](#)
JmxCamelClient class [700](#)
JmxNotificationEventNotifier [692](#)
JNDI (Java Naming and Directory Interface) registry [19](#)
JndiRegistry [114–115, 216](#)
jobs, scheduling with Quartz [79](#)
Johnson, Ralph [78](#)
Jolokia agent
 as ping service [673–675](#)
 managing application lifecycles with [695–697](#)
 managing Camel with [680–684](#)
 using Java agents [683–684](#)
 using with Apache Karaf [682](#)

WAR agents [681–684](#)

JPA (Java Persistence API) components [218–226](#)

JpaEndpoint [226](#)

JpaMessageIdRepository [556](#)

JpaTransactionManager [222](#)

jps tool [731](#)

jsch dependency [312](#)

json_xml mode [434](#)

JSON (JavaScript Object Notation)

- adding support to camel-cxf [424–425](#)
 - to/from POJOs using camel-jackson [419–420](#)
 - with Rest DSL [436–441](#)
- data formats [97, 434–443](#)
- instead of XML [413–414](#)
- marshaling beans [97](#)

jsonDataFormat option [436](#)

json mode [434](#)

JSSE (Java Secure Socket Extension) [614](#)

JTA (Java Transaction API) [531](#)

JtaTransactionManager [531, 533, 536](#)

JUnit extensions [346](#)

JVisualVM [676](#)

JVM (Java Virtual Machine)

- checking health at level of [675](#)
- connecting hawtio to [820–821](#)
- crashing with running Kafka consumers [731–733](#)

using properties [679](#)
JVM level health checks [672](#)
JXPath [56](#)

K

kafka.setBrokers [728](#)
KafkaManualCommit API [731](#)
Kafka messaging system
 clustered [727–733](#)
 consumer offsets [730–731](#)
 crashing JVM with running consumers [731–733](#)
Karaf [4](#)
keepAliveTime option [572, 573](#)
keyManagers [615](#)
key pair [609](#)
KeyStoreParameters [611–613](#)
key-value pairs [779](#)
killing pods [785](#)
kind option, JSON schema [816](#)
Korab, Jakub [732](#)
kubectl CLI, running applications using [765–766](#)
kubectl get pods command [782](#)
Kubernetes [244](#)
KubernetesClient [793, 795](#)
Kubernetes platform [776–783](#)
 architecture of [778](#)

basic concepts [778–783](#)
 deployments [780](#)
 labels [779](#)
 pods [779](#)
 replication controllers [780](#)
 services [781–783](#)

building microservices on [783–792](#)
dealing with failures by calling services in [789–792](#)
manifest file [767–768](#)
microservices with [752–764](#), [776–801](#)
 calling services running inside Kubernetes clusters
[766–768](#)
 debugging Java applications in Kubernetes [774–776](#)
 installing Minikube [762](#)
 Java microservices calling each other in clusters [771–773](#)
 running applications using kubectl [765–766](#)
 running Java applications in Kubernetes using Maven
 tooling [768–770](#)
 starting Minikube [763–764](#)
Minikube interacting with [763–764](#)
 origin of [777](#)
 overview [776–777](#)
 platforms built on top of [798](#)
 scaling up microservices [784–786](#)
 deleting pods [784](#)
 killing pods [785](#)
 scaling up pods [786](#)

testing microservices on [792–795](#)
 running Arquillian Cube tests on Kubernetes [793–794](#)
 setting up Arquillian Cube [792](#)
 writing unit tests calling Kubernetes services [794–795](#)
 writing unit tests using Arquillian Cube [793](#)
 using Docker CLI with [764](#)
 using liveness probes [786–789](#)
 using readiness probes [786–789](#)

L

labels [779](#)
Label selector, replication controller [780](#)
language annotations, binding parameters with [132–136](#)
 using @JsonPath binding annotation [135–136](#)
 using namespaces with @XPath [135](#)
Language component [198](#)
Least Recently Used. See LRU (Least Recently Used)
LevelDBAggregationRepository [159–165](#)
LevelDB library [160–161](#)
LGPL license [654](#)
lib directory [10](#)
LICENSE.txt [10](#)
lifecycle management [247](#)
lipRoutePolicy [640](#)
listening to events, with CDI [251–252](#)
-list keytool command [609](#)

literal values [138](#)
liveness probes [786–789](#)
Load Balancer EIPs (enterprise integration patterns) [147](#),
[184–193](#)
 across remote service [184](#)
 introducing [186](#)
 overview [148](#), [184–186](#)
 strategy [185](#), [187](#)
 custom [186](#)
 failover [190](#)
 failover by exception [188](#)
 failover inherit error handler [189](#)
 failover maximum attempts [189](#)
 failover with round robin [190](#)
 random [186](#)
 round robin [185](#), [186](#)
 sticky [186](#), [187](#)
 topic [186](#)
 using custom load balancers [190–193](#)
 using failover load balancer [188–190](#)
 using load-balancing strategies [186–187](#)
load balancers
 custom [190–193](#)
 failover [188–190](#)
 using strategies [186–187](#)
LoadBalancer type [769](#)

loaded data formats [19](#)
loaded type converters [19](#)
loading type converters, into registry [101–102](#)
load-on-startup option [633](#)
localhost broker [32](#)
locally managed transactions [521](#)
Local option [677](#)
local transactions [530–531, 560](#)
log://DLC endpoint [489](#)
log:display command [662](#)
log:tail command [662](#)
Log component [198](#)
Log EIP [688–689](#)
LogExhaustedMessageBody option [629](#)
LogExhaustedMessageHistory option [483](#)
LogExhausted option [483](#)
log files [685](#)
logging [685–689](#)
 using Camel Log components [687–688](#)
 using correlation IDs [689](#)
 using Log EIP [688–689](#)
 using wire tap for [686–687](#)
LoggingErrorHandler [474, 481](#)
LoggingEventNotifier [691, 692](#)
log method, DSL [57](#)
LogRetryAttempted option [483](#)

logs [316](#)
LogStackTrace option [483](#)
lookup method [101](#)
loose coupling [145](#)
lost messages [519–521](#)
lost update problem [735](#)
LowPool [584](#)
LRU (Least Recently Used) [556](#)
Luksa, Marko [801](#)

M

m2eclipse plugin, running a project [314](#), [315](#)
Mail component, email [234](#)
mailing lists, for Camel community [835–836](#)
Main class [246](#), [247](#), [248](#), [353](#), [647](#), [712](#)
main method [722](#)
ManagedBacklogDebugger MBean [820](#)
ManagedCamelContextMBean interface [698](#), [699](#)
ManagedPerformanceCounterMBean [705](#)
ManagedRoute MBean [635](#)
managed service factory. *See* MSF (managed service factory)
ManagedThrottlerMBean interface [703](#)
Management activity [572](#)
ManagementAgent interface [678](#), [679](#)
management-enabled components [708–711](#)
management-enabled Java beans [711–712](#)

ManagementStrategy [692](#)
managing projects with Maven [307–313](#)
 adding dependencies [311–313](#)
 using archetypes [307–311](#)
MANIFEST.MF entry [658, 659](#)
man-in-the-middle attack [611](#)
manipulating routes, with adviceWith method [383–387](#)
mapJmsMessage URI option [212](#)
MapMessage [211](#)
mapping
 data [77](#)
 messages [210–212](#)
 body mapping [210–212](#)
 header mapping [212](#)
markRollbackOnly attribute [543](#)
marshal DSL method [613](#)
marshaling, transforming XML with [88–91](#)
 JAXB [90–91](#)
 transforming using XStream [89](#)
marshal method [91, 338, 340](#)
Martin, Robert C. [242](#)
matches method [56, 373, 392, 394](#)
Maven archetype plugin [307](#)
maven-bundle-plugin [659](#)
Maven project manager
 managing projects with [307–313](#)

setting up to generate OSGi bundle [659](#)
using archetypes [307–311](#)
validating Camel using [811–813](#)

Max-Age option [463](#)

maxDeep method [389](#)

maximumFailoverAttempts option [189, 190](#)

maximumPoolSize option [572](#)

maximumRedeliveries option [166, 493, 496, 497](#)

MaximumRedeliveryDelay option [483](#)

maxMessagesPerPoll option [636](#)

maxPoolSize option [573](#)

maxQueueSize option [288, 289, 573, 574, 580](#)

MBeans (management beans) [677](#)

MBeanServiceConnection instance [701](#)

mediation engines [5](#)

MemoryAggregationRepository [159](#)

MemoryIdempotentRepository [555](#)

MEPs (message exchange patterns) [16, 209](#)

messageConverter option [212](#)

message filters [61–62](#)

MessageHistory option [629](#)

messageIdRepositoryRef attribute [554](#)

message key, Kafka [728](#)

message method [373](#)

Message parameter [130](#)

Message Queue Telemetry Transport. *See* MQTT (Message

Queue Telemetry Transport)

messages

aggregating [174–175](#)

asynchronous [228–230](#)

attachments [16](#)

binding negotiation for [435–436](#)

body [16](#)

enriching with cause of error [478–480](#)

exchange [16–18](#)

fault flags [16](#)

headers [16](#)

in-flight [641, 642, 645](#)

lost [519–521](#)

mapping [210–212](#)

body mapping [210–212](#)

header mapping [212](#)

processing [392](#)

receiving [208–209](#)

reply, constructing [543–544](#)

request-reply [209–210](#)

sending [208–209](#)

splitting [172–173](#)

synchronous [227](#)

tracing [801](#)

verifying arrival of [371–372](#)

Message Translator EIPs, transforming data with [78–83](#)

using <transform> from XML DSL [83](#)
using beans [80–81](#)
using processors [78–80](#)
using transform method from Java DSL [81–83](#)
messaging [727](#)
META-INF directory [104](#)
method element [67](#)
method facades [145](#)
method names [110](#), [145](#)
MethodNotFoundException [120](#), [123](#), [125](#), [126](#)
methodPattern element [332](#)
methods
 bean, selecting [119–120](#), [121–128](#)
 method-selection algorithms [121–123](#)
 method-selection examples [123–125](#)
 method-selection problems [125–127](#)
 selecting using type matching [127–128](#)
 using names with signatures [136–138](#)
 method-selection
 algorithms [121–123](#)
 examples of [123–125](#)
 troubleshooting [125–127](#)
 method signatures [81](#), [97](#), [102](#), [145](#)
 <method> tag [172](#)
MetricsRegistryService [707](#)
metricsRollingStatisticalWindowInMilliseconds option [288](#)

MetricsRoutePolicyFactory [706](#)
microservices
 building and running locally [754–755](#)
 building and running using Docker [756–759](#)
 building on Kubernetes [783–792](#)
 dealing with failures by calling services in Kubernetes
 [789–792](#)
 scaling up microservices [784–786](#)
 using liveness probes [786–789](#)
 using readiness probes [786–789](#)
 calling [262–276](#)
 inventory prototype [270–273](#)
 overview [275–276](#)
 rating prototype [273–274](#)
 recommendation prototype [264–266](#)
 rules prototype [270–273](#)
 shopping cart prototype [266–269](#)
 technologies [263–264](#)
 with fault-tolerance [292–298](#)
 CDI Camel as [247–252](#)
 configuring applications [248–249](#)
 dependency injection using @Inject [249–250](#)
 hello service with [247–248](#)
 listening to events using [251–252](#)
 overview [252](#)
 using @Uri to inject endpoints [250–251](#)

designing for failures [277–305](#)

- bulkhead patterns [289–292](#)
- calling microservices with fault-tolerance [292–298](#)
- using Circuit Breaker [280–282](#)
- using Hystrix [282–288, 301–305](#)
- using Hystrix with Camel [284–286](#)
- using Hystrix with Spring Boot [298–301](#)
- using Retry patterns [277–280](#)

lightweight core for [9](#)

overview [242–245](#)

- configurable [244](#)
- designed for failure [242](#)
- observable [242](#)
- size of [242](#)
- smart endpoints and dumb pipes [244](#)
- testable [245](#)

running [245–262](#)

Spring Boot with Camel as [256–262](#)

- adding Camel to existing Spring Boot REST endpoints [258](#)
- adding REST to Spring Boot applications [257–258](#)
- monitoring Camel with Spring Boot [262](#)

overview [256–257, 262](#)

using property placeholders with Spring Boot [261](#)

using routes with Spring Boot [259–260](#)

using starter components with Spring Boot [259](#)

using XML DSL with Spring Boot [260–261](#)

standalone Camel as [245–247](#)

- overview [247](#)
- running by using main classes [246](#)

testing on Kubernetes [792–795](#)

- running Arquillian Cube tests on Kubernetes [793–794](#)
- setting up Arquillian Cube [792](#)
- writing unit tests calling Kubernetes services [794–795](#)
- writing unit tests using Arquillian Cube [793](#)

using fabric8 [796–800](#)

using Helm [796–800](#)

- installing [797](#)
- installing applications using Helm [797–798](#)
- platforms built on top of Kubernetes [798](#)

using OpenShift [796–800](#)

WildFly Swarm with Camel as [252–256](#)

- fragments versus components [254–255](#)
- monitoring Camel with WildFly Swarm [255–256](#)
- routes with Spring XML [255](#)
- with C for microservices [256](#)

building images using Docker Maven plugins [758–759](#)

- running Java microservices on [759–762](#)

with Kubernetes [752–762, 764, 776, 783–801](#)

- architecture of [778](#)
- basic concepts [778–783](#)
- calling services running inside Kubernetes clusters

766–768

- debugging Java applications in Kubernetes 774–776
- installing Minikube 762
- Java microservices calling each other in clusters 771–773
- overview 776–777
- running applications using kubectl 765–766
 - running Java applications in Kubernetes using Maven tooling 768–770
 - starting Minikube 763–764
- migrating applications, to OSGI bundles 400–401
- minikube start command 764
- minikube stop command 764
- Minikube tool
 - installing 762
 - starting 763–764
 - interacting with Kubernetes 763–764
 - using Docker CLI with Kubernetes 764
- mock:encrypted endpoint 614
- mock:miranda endpoint 377
- mock:quote endpoint 370, 373
- mock:unencrypted endpoint 614
- MockAuditService 361
- mock components 368–378
 - overview 369
 - simulating errors with 380–381
 - to simulate real components 376–378

unit testing with [369–371](#)
using expressions with [372–376](#)
using mock components to simulate [376–378](#)
verifying arrival of correct messages [371–372](#)

mock endpoint [168](#)
MockEndpoint class [96, 370](#)
mockEndpoints method [384](#)
mocking OSGI services [360–361](#)
modeling messages [15–18](#)
monitor fraction [255](#)
monitoring [672–676](#)
 checking health at application levels [675–676](#)
 checking health at JVM levels [675](#)
 checking health at network levels [672–675](#)
 retries [279](#)
 with Spring Boot [262](#)
 with Wildfly Swarm [255–256](#)

Moulliard, Charles [94](#)

move option [201](#)

MQTT (Message Queue Telemetry Transport) [244](#)

MSF (managed service factory) [663–665](#)

Multicast EIP, parallel processing [581–583](#)

multicasting [62–65](#)
 parallel, using [64](#)
 stopping on exceptions [64–65](#)

multicast method [63](#)

multitasking [562](#)
Mvel [56](#)
MVEL (MVFLEX Expression Language) [83](#)
mvn exec:java command [315](#)
MyApplication class [353](#), [357](#)
MyCommand class [282](#)
MyComponent class [320](#), [323](#)
MyConsumer class [324](#)
myDest header [50](#)
MyEndpoint class [323](#)
MyHttpUtil bean [510](#)
myLock [723](#)
#myPolicy value [722](#)
MyRetryRuleset class [512](#)
mySlip key [179](#)

N

Nagios [672](#)
nameDefinition pattern [389](#)
NamespaceHandler [627](#)
namespaces, using with @XPath [135](#)
Narayana [532](#)
Narkhede, Neha [732](#)
negotiation, binding for messages [435–436](#)
Netflix Hystrix
calling commands from Java [283](#)

configuring [287–288](#)
 options [288](#)
 with Camel [287–288](#)
 with Java [287](#)

dashboard [301–305](#)

fallback with [283–284, 290–291](#)

time-outs with [289–290–291](#)

with Camel [284–286, 296–297](#)
 adding fallbacks [285–286](#)
 configuring command keys [286](#)

with plain Java [282](#)

with Spring Boot [293–295, 298–301](#)

with Wildfly Swarm [296–297](#)

Netty4 components [213–217](#)
 for network programming [213–215](#)
 using custom codecs [216–217](#)

networking [195](#)

Network level health checks [672](#)

networks

 checking health at level of [672–675](#)
 fallback via [292](#)
 programming with Netty4 components [213–215](#)

newBody message [158](#)

newBookOrders [508](#)

newCachedThreadPool method [567](#)

newExchange parameter [85, 177](#)

Newman, Sam [242](#)

noAutoStartup() method [635](#)

NodePort type

configuring in pom.xml [770](#)

overview [766, 769, 775](#)

nodes, selecting with weave [389–390](#)

NoErrorHandler [474, 480](#)

nonblocking mode [563](#)

noop option [13, 44](#)

noStreamCaching method [630](#)

NotAllowedProcessor [502](#)

NOTICE.txt [10](#)

notifications

configuring event notifiers [692](#)

using custom event notifiers [692–694](#)

NotifyBuilder [348, 392, 393–395, 528](#)

noTracing method [630](#)

NoTypeConversionAvailableException [103, 125](#)

nullableOptions [332](#)

NullPointerException [502](#)

null values

overview [138](#)

returning [104](#)

numeric values [138](#)

Nygard, Michael [280](#)

O

ObjectFactory class [440](#)

Object-Graph Navigation Language. *See* OGNL (Object-Graph Navigation Language)

object ID [734](#)

object marshaling, transforming XML with [88–91](#)

JAXB [90–91](#)

transforming using XStream [89](#)

ObjectMessage [211](#)

object-relational mapping. *See* ORM (object-relational mapping)

objects, serialization codecs [215](#)

objects persisting with JPA components [221–226](#)

off mode [434](#)

OGNL (Object-Graph Navigation Language) [56](#)

accessing bean [832](#)

accessing List [833](#)

accessing Map [833](#)

bean parameter binding [833](#)

in Simple [832–833](#)

null safe operator [833](#)

OkHttpClient library [795](#)

oldBody message [158](#)

oldExchange parameter [85, 152, 177](#)

onComplete method [545](#)

OnCompleteOnly option [637](#)

onCompletion

BeforeConsumer mode [550–551](#)

predicates with [549](#)

starting and stopping routes at runtime [636–638](#)

onException method [492](#)

catching exceptions with [492–495](#)

handling exceptions with [497–498–500](#)

doCatch [498](#)

doFinally [498](#)

doTry [498](#)

multiple exceptions per [495](#)

redelivery and [496–497](#)

onExceptionOccurred feature [510–511](#)

OnExceptionTest class [493](#)

onExecutionOccurred [509](#)

onFailure method [545, 546](#)

onFailureOnly attribute [548](#)

OnFailureOnly option [637](#)

onPrepareFailureRef attribute [479](#)

onRedeliver feature [509, 511](#)

onWhen feature [509–510](#)

OnWhen predicate [637](#)

OpenJPA [221](#)

Open Service Gateway initiative. See OSGi (Open Service Gateway initiative) framework

OpenShift platform [796–798, 800](#)

open source tools [672](#)
OpenTracing [801](#)
operators, in Simple [830–832](#)
optimisticLocking option [157](#)
optimisticLockRetryPolicy option [157](#)
orderAudit queue [70](#)
OrderCsvToXmlBean bean [686, 687](#)
OrderEndpoint [603](#)
orderEndpoint CXF endpoint [227](#)
OrderFailedException [492, 493, 495](#)
orderId attribute [134](#)
OrderInvalidException [441, 457](#)
order messages, sending using clients [391](#)
OrderNotFoundException [441, 457](#)
Order object [600](#)
OrderQueryProcessor [376](#)
OrderResponseProcessor [376](#)
OrderResult [600](#)
OrderRouteBuilder class [492](#)
OrderRoute class [442](#)
OrderRouterWithFilterTest.java [62](#)
orderService bean [97, 486](#)
OrderServiceBean class [126](#)
OrderToCsvBean [81](#)
orderToSql bean [220](#)
org.apache.camel.spi.DataFormat [338](#)

org.apache.camel [254](#)
org.apache.camel.builder package [393](#)
org.apache.camel.cdi.Main class [248](#)
org.apache.camel.component.mock.MockEndpoint class [370](#)
org.apache.camel.core package [104](#)
org.apache.camel.Endpoint [250](#)
org.apache.camel.Exchange [15](#), [471](#), [472](#)
org.apache.camel.Expression instance [168](#)
org.apache.camel.main.Main class [25](#), [246](#)
org.apache.camel.Message [15](#)
org.apache.camel.Processor [78](#), [511](#)
org.apache.camel.processor.LoggingErrorHandler [481](#)
org.apache.camel.processor.RedeliveryErrorHandler [481](#)
org.apache.camel.spi.Registry interface [113](#)
org.apache.camel.test.AvailablePortFinder class [346](#)
org.apache.camel.test.junit4.CamelTestSupport class [346](#)
org.apache.camel.test.junit4.TestSupport class [346](#)
org.wildfly.swarm [254](#)
ORM (object-relational mapping) [221](#)
OSGi (Open Service Gateway initiative) framework [658–665](#)
 Blueprint XML [358–361](#)
 bundles [400–401](#)
 deploying example [661–662](#)
 hot deployment [658](#)
 import and export [659](#)
 installing and running Apache Karaf [660](#)

Maven bundle plugin [659](#)
services [360–361](#)
setting up Maven to generate OSGi bundle [659](#)
testing with [349–368](#)
using Blueprint-based Camel route [661](#)
using MSF to spin up route instances [663–665](#)
OsgiServiceRegistry [114](#), [116–117](#)
otherwise block [61](#)
otherwise clause [58–59](#)
otherwise method [59](#)
outgoing messages [434](#), [435](#), [436](#), [438](#)
output, documenting [454–459](#)
 body output types [455–456](#)
 header output types [457](#)
 output types [455](#)
output messages [16](#)

P

PaaS (platforms as a service) [798](#)
package command [650](#)
packageScan element [46](#)
parallel multicasting [64](#)
parallel processing [562–594](#)
 asynchronous routing engine [586–593](#)
 asynchronous API [590–591](#)
 components supporting asynchronous processing [589](#)

hitting scalability limit [586–587](#)
potential issues [593](#)
scalability in Camel [588–589](#)
writing custom components [591–593](#)

concurrency [563–586](#)
EIPs [578–586](#)
enabling parallelProcessing() [566–567](#)
ExecutorServiceManager [576–578](#)
running example without [565](#)
SEDA [568–570](#)
thread pools [567–568, 571–576](#)

parallelProcessing() method
enabling [566–567](#)
overview [64](#)

parameters
binding [128–138](#)
using annotations [130–132](#)
using built-in types [130](#)
using language annotations [132–136](#)
using method names with signatures [136–138](#)
with multiple parameters [129–130](#)
working with [130](#)

partIds method [272](#)
passive mode [718](#)
Pauls, Karl [358](#)
Pax Exam

adding artifacts [402](#)
setting up [402](#)
testing applications with [399–404](#)
 migrating applications to OSGI bundles [400–401](#)
 running tests [404](#)
 writing integration test classes [402–403](#)
payload-agnostic routers [8](#)
payload conversion [38](#)
payload security [608–614](#)
 digital signatures [608–612](#)
 generating and loading public and private keys [609–610](#)
 signing and verifying messages using camel-crypto
[610–612](#)
 payload encryption [612–614](#)
 encrypting and decrypting payloads using camel-crypto
[613–614](#)
 generating and loading secret keys [612–613](#)
performance statistics, of Camel management API [705–711](#)
period (.) characters [212](#)
period option, JSON schema [816](#)
persistence, using with Aggregator EIPs [159–161](#), [163–](#)
persistence.xml file [225](#)
persisting objects, with JPA components [221–226](#)
PetSet [783](#)
PGPDataformat [612](#)
PID (persistent identifier) [665](#), [731](#)
ping services, Jolokia as [673–675](#)

`pipeline` 37

`-P kubernetes` argument 793

placeholders 50, 757

 using in endpoint URIs 50–53

 using in Spring 52–53

 using in XML DSL 51–52

platforms, building on top of Kubernetes 798

platforms as a service. *See PaaS* (platforms as a service)

plugins, adding to projects 812–813

plus characters 233

PMC (Project Management Committee) 837

pods

 debugging 774–775

 deleting 784

 killing 785

 scaling 786

 scaling up with liveness probes 788–789

 scaling up with readiness probes 788–789

PodStarterKT class 793

Pod template, replication controller 780

POJOs (Plain Old Java Objects)

 binding JSON to/from using camel-jackson 419–420

 binding XML to/from using camel-jaxb 419

 using for AggregationStrategy 158–159

PojoSR 358

Policy class 618

`pollEnrich`

enriching data with [85–87](#)

vs. `enrich` [85](#)

polling, consumers [24](#)

`pom.xml`, configuring NodePort in [770](#)

POM (Project Object Model [13](#))

`poolSize` option [573](#)

`port` option [430](#)

ports, calling service in Kubernetes changing port numbers
[775–776](#)

Posta, Christian [242](#), [801](#)

POST verb [410](#)

predicate interface [56](#)

predicates

compound, using in Java [141](#)

in routes, using beans as [138–145](#)

`onCompletion` with [549](#)

`PreparedStatement` [220](#)

`prettyPrint` option [436](#)

previous parameter [182](#)

processing messages [392](#)

process method [108](#), [191](#), [324](#), [590](#), [592](#)

Processor API [77](#)

processors [21–22](#)

adding in Java DSL [38–39](#)

adding in XML DSL [44–45](#)

simulating errors with [378–380](#)
to transform data [78–80](#)
producers [23, 38, 323–326](#)
ProducerTemplate class [181, 227, 258, 345, 518](#)
produces option [436](#)
programming, networks with Netty4 components [213–215](#)
Project Management Committee. *See* PMC (Project Management Committee)
projects [306–342](#)
 adding plugins to [812–813](#)
 data format [338–339](#)
 debugging [316–318](#)
 developing custom components [318–326](#)
 implementation of [320–326](#)
 setting up new components [318–319](#)
 developing data formats [338–342](#)
 generating data format projects [338–339](#)
 writing custom data formats [339–342](#)
 generating components with API component framework
[326–337](#)
 configuring camel-api-component-maven-plugin
[328–330](#)
 generating skeleton API project [326–328](#)
 implementing [335–337](#)
 setting advanced configuration options [331–335](#)
 managing with Maven [307–313](#)
 adding dependencies [311–313](#)

using Maven archetypes [307–311](#)
starting new [313–315](#)
using Camel in Eclipse [313–315](#)
project templates [307](#)

Prometheus [672](#)
promptMessage option [201](#)
propagating, transactions [539–541](#)
PROPAGATION_REQUIRED [539](#)
propagation scope [561](#)
PropertiesComponent [248, 598](#)
Properties component [198](#)
propertiesParserRef attribute [597](#)
PropertyParser [598](#)
property placeholders
 using in endpoint URIs [50–53](#)
 using in Spring [52–53](#)
 using in XML DSL [51–52](#)
 with Spring Boot [261](#)
<property> tag [508](#)
Protocol Buffers [244](#)
providing data formats [92](#)
PRs (pull requests) [836](#)
pseudocode [497](#)
public and private keys, generating and loading [609–610](#)
publish/subscribe scheme [31](#)
PublishEventNotifier [691, 692](#)

PurchaseOrder object [90](#), [91](#)

PUT verb [410](#)

Q

quartz.properties file [738](#)

Quartz2 components [230](#)–[233](#)

Quartz components

- Camel routes using [738](#)–[739](#)

- clustered scheduling using [737](#)–[739](#)

- configuring to use databases [738](#)

- scheduling jobs with [79](#)

- setting up databases for [738](#)

queues

- inspecting [392](#)–[393](#)

- overview [31](#)

R

Random strategy [186](#)

RatingDto.java file [264](#)

rating prototype [273](#)–[274](#)

rating services [274](#)

raw values, using in endpoint URIs [54](#)

readiness probes, scaling up pods with [786](#)–[789](#)

reading files, with File components [200](#)–[201](#)–[203](#)

README.txt [11](#)

receiving

emails with IMAP [235–237](#)
messages [208–209](#)

Recipient List EIP [66, 578](#)
recipientList method [66, 67](#)

Recipient List pattern [141, 143](#)
recipient lists [66–69](#)

RecipientsBean class [67](#)
recommendation prototype [264–266](#)

RecommendController.java file [264](#)
RecommendController class [264, 294](#)

RecoverableAggregationRepository [163, 164](#)
recoverable errors [470–472](#)

recovery, using with Aggregator EIPs [163–165, 166](#)
recoveryInterval option [166](#)

redelivery
 broker-level with ActiveMQ [538](#)
 consumer-level with ActiveMQ [537](#)
 DefaultErrorHandler with [483–487](#)
 error handlers with [481–482, 492](#)
 of transactions [536–539](#)
 onException and [496–497](#)
 using [482–483](#)

RedeliveryDelay option [483](#)
RedeliveryErrorHandler class [474, 482](#)
<redeliveryPolicy> element [483](#)

Redis [733](#)

ref attribute [67](#)

Ref component [198](#)

refNo field [544](#)

regex method [373](#)

regexReplaceAll [82](#)

register method [354](#)

registries [30](#)

loading type converters into [101–102](#)

of beans [113–119](#)

ApplicationContextRegistry [116](#)

BlueprintContainerRegistry [116–117](#)

CdiBeanRegistry [117–119](#)

JndiRegistry [114–115](#)

OsgiServiceRegistry [116–117](#)

referencing in endpoint URIs [54](#)

SimpleRegistry [115–116](#)

Registry parameter [130](#)

registryPort attribute [679](#)

RejectedExecutionException [740, 741](#)

RejectedExecution-Handler type [572](#)

rejected option [572](#)

rejectedPolicy option [573](#)

Remote Method Invocation [676](#)

remove method [553](#)

removeOnFailure option [553](#)

removeSynchronization method [545](#)

ReplaceFromTest class [386](#)
replacement element [331](#)
replacing
 JMS with stubs [371](#)
 route endpoints [385–387](#)
Replica count, replication controller [780](#)
replication controllers [780](#)
replyError method [543](#)
reply messages, constructing [543–544](#)
repositories, idempotent [555–556](#)
 clustered [556–561](#)
 FileIdempotentRepository [555](#)
 HazelcastIdempotentRepository [556](#)
 InfinispanIdempotentRepository [556](#)
 JdbcMessageIdRepository [555](#)
 JpaMessageIdRepository [556](#)
 MemoryIdempotentRepository [555](#)
requestBody method [377](#)
request-reply messaging [209–210](#)
resourceClasses option [421](#)
resources, for Camel community [838](#)
responses, returning when transactions fail [542–544](#)
 catching exceptions [542–543](#)
 constructing reply messages [543–544](#)
 handling exceptions [542–543](#)
 rolling back transactions [543](#)

RestBindingProcessor 434, 435, 438
REST component 198
 adding to Spring Boot applications 257–258
 endpoints 258
rest-dsl 550
Rest DSL (domain-specific language) 425–444
 calling RESTful services with 443–444
 configuring 429–434
 binding 436
 components 431–434
 consumer options 431–434
 endpoints 431–434
 options 430–431
 handling exceptions with 441–443
 JSON binding with 436–439–441
 JSON data formats with 434–443
 overview 427–428
 returning custom HTTP status codes with 441–443
 Spring Boot configuration with 441
 supported components for 429
 XML binding with 439–441
 XML data formats with 434–443
RESTful web services 408–409, 425–466
 API documentation using Swagger 444–465
 configuring API documentation 459–461
 documenting error codes 454–457, 459

documenting input [454–459](#)
documenting output [454–459](#)
documenting Rest DSL services [452–453](#)
Swagger overview [445–446](#)
Swagger with JAX-RS REST services [446–451](#)
Swagger with Rest DSL [451–452](#)
using CORS [461–465](#)
using Swagger web console [461–465](#)
camel-cxf with [420–425](#)
 adding JSON support to [424–425](#)
Camel-configured endpoints [420–422](#)
 CXF-configured beans [422–424](#)
camel-restlet with [417–420](#)
 binding JSON to/from POJOs using camel-jackson
[419–420](#)
 binding XML to/from POJOs using camel-jaxb [419](#)
documenting [452–453](#)
JAX-RS with [410–414](#)
 using Camel in existing applications [414–417](#)
 using JSON instead of XML [413–414](#)
Rest DSL [425–444](#)
 calling with [443–444](#)
 configuring [429–434](#)
 overview [427–428](#)
 supported components for [429](#)
 using JSON data formats with [434–443](#)

using XML data formats with [434–443](#)

RESTful API [409–410](#)

restHostNameResolver option [430](#)

restlet component [418](#), [419](#)

restletMethod option [418](#)

RestOrderApplication class [449](#)

RestTemplate [265](#)

Retry patterns

- handling failures with [277–279](#)
 - how often to retry [278](#)
 - idempotency [278–279](#)
 - monitoring [279](#)
 - SLAs [279](#)
 - time-outs [279](#)
 - when to use [278](#)
- without Camel [279–280](#)

retryWhile feature [509](#), [512](#)

reusing context-scoped error handlers [490–492](#)

Rider Auto Parts

- inventory updates from suppliers [626](#)
- web service for order submission [600](#)

RiderAutoPartsPartnerTransactedTest class [526](#)

RiderEventNotifier [693](#)

RiderFailurePublisher class [693](#)

rider-test.properties file [51](#)

RoleVoter [620](#)

RollbackExchangeException 543
rollback strategies 543
rolling back, exceptions 543
round-robin strategy
 overview 185, 186
 using with failover load balancers 189–190
route
 additional routing using OnCompletion 636
 AutoStartup 631
 difference between stop and suspend 645
 flip routes being active 639
 StartupOrder 631
route authentication and authorization 618–621
RouteBuilder
 adding to CamelContext 34
 anonymous RouteBuilder class 34
 configure method 34
 testing with only one enabled 357–358
 unit testing 349
route builders, finding 46–47
routeContext element 49
route debugger 821
RouteDefinition 35
route element 44
routeId 632
route method 182

route parameter [640](#)

RoutePolicy, starting and stopping routes at runtime [639–641](#)

RoutePolicy class [482](#)

routePolicyRef attribute [641](#)

RoutePolicySupport class [639, 640](#)

routers, payload-agnostic [8](#)

routes

- adding interceptors with adviceWith method [382–383](#)
- advising [384–385](#)
- amending using weave with adviceWith method [387–390](#)
 - selecting nodes with weave [389–390](#)
 - weave without using IDs [388–389](#)
- clustered [720–726](#)
 - active/active mode [720–721](#)
 - active/passive mode [720](#)
 - active/passive mode using Consul [723–724](#)
 - active/passive mode using Hazelcast [721–723](#)
 - active/passive mode using ZooKeeper [724–726](#)
- creating in Java [33–39](#)
 - using DSLs [35–39](#)
 - using RouteBuilder [34–35](#)
- debugging using hawtio [818–821](#)
- defining in XML [39–49](#)
 - creating applications from beans using Spring [40–43](#)
 - running Camel in Spring containers [46–49](#)
- XML DSLs [43–45](#)

endpoints, replacing [385–387](#)
filtering in API documentation [460–461](#)
importing [48–49](#)
introduction [12](#)
Java Camel, testing with camel-test [346–349](#)
ManagedRoute MBean [635](#)
manipulating with adviceWith method [383–387](#)
ordering [631–634](#)
 how StartupOrder works [633–634](#)
 using StartupOrder to control [631–633](#)
ordering example [631](#)
reverse order [633](#)
starting and stopping at runtime [635–641](#)
 using CamelContext [635–638](#)
 using Control Bus EIP [638](#)
 using RoutePolicy [639–641](#)
starting and stopping via JMX [635](#)
StartupOrder [632](#)
 using beans as expressions in [138–145](#)
 using beans as predicates in [138–145](#)
 using MSF to spin up route instances [663–665](#)
 using multiple [45](#)
 using Quartz [738–739](#)
 with Spring Boot [259–260](#)
 with Spring XML [255](#)
<route> tag [641](#)

route trace [486](#)

routing [27–71](#)

- after CBRs [60–61](#)

- creating routes in Java [33–39](#)

 - using DSLs [35–39](#)

 - using RouteBuilder [34–35](#)

- defining routes in XML [39–49](#)

 - creating applications from beans using Spring [40–43](#)

 - running Camel in Spring containers [46–49](#)

 - XML DSLs [43–45](#)

- EIPs and [55–71](#)

 - using CBR [55–61](#)

 - using message filters [61–62](#)

 - using multicasting [62–65](#)

 - using recipient lists [66–69](#)

 - using wireTap method [69–71](#)

- endpoints [29–33, 49–54](#)

 - consuming from FTP endpoints [30–31](#)

 - sending to dynamic endpoints [49–50](#)

 - sending to JMS endpoints [31–33](#)

 - using property placeholders in endpoint URIs [50–53](#)

- routing engine [586–593](#)

 - asynchronous API [590–591](#)

 - components supporting asynchronous processing [589](#)

 - hitting scalability limit [586–587](#)

 - overview [5, 20](#)

potential issues [593](#)
scalability in Camel [588–589](#)
writing custom components [591–593](#)

Routing Slip EIPs

headers, computing with beans [179–180](#)
overview [148, 179](#)
using @RoutingSlip annotations [180–182](#)
using beans to compute routing slip headers [179–180](#)
using Expressions as [180](#)
using Expressions as routing slips [180](#)
using header as slip [178](#)
rsServer element [422, 423](#)
RulesController class [271](#)
rules microservices [271–272](#)
rules prototype [270–273](#)
 rules microservices [271–272](#)
 running rules services [273](#)
 rules services [273](#)
run command [758, 765, 767](#)
run method [246, 282, 283, 284](#)
RuntimeCamelException [492](#)

S

scalability
 hitting limit [586–587](#)
 in Camel [588–589](#)

scaling

 microservices [784–786](#)

 deleting pods [784](#)

 killing pods [785](#)

 with liveness probes [788–789](#)

 with readiness probes [788–789](#)

 scheduledExecutorService [230](#)

 ScheduledPollConsumer [325](#)

 Scheduler components [198](#), [230](#), [231–233](#)

 scheduleWithFixedDelay method [577](#)

 scheduling, clustered [737–739](#)

 scheme option [430](#)

 scopes, error handlers and [487–490](#)

 scripting language, using as expression [83](#)

 secret keys, generating and loading [612–613](#)

 secret option, JSON schema [816](#)

 Secure Sockets Layer. *See* SSL (Secure Sockets Layer)

 security [595–621](#)

 configuration security [596–599](#)

 decrypting configuration [597–599](#)

 encrypting configuration [596–597](#)

 payload security [608–614](#)

 digital signatures [608–612](#)

 payload encryption [612–614](#)

 route authentication and authorization [618–621](#)

 transport security [614–617](#)

web service security [599–607](#)

seda:outbox endpoint [364](#)

SEDA (staged event-driven architecture) [226, 566, 568–570](#)

SEDA components

- asynchronous messaging with [228–230](#)
- overview [198, 226–230](#)
- selectFirst method [389](#)
- selectIndex method [389](#)
- selecting nodes with weave [389–390](#)
- selectLast method [389](#)
- selectRange method [389](#)
- sending
 - emails with SMTP [234–235](#)
 - messages [208–209](#)
 - order messages using clients [391](#)
 - to dynamic endpoints [49–50](#)
- sendOrder method [391](#)
- Serializable Java object [215](#)
- serializing objects, codecs for [215](#)
- ServerBar class [721, 736](#)
- ServerFoo class [721, 736](#)
- ServerPasswordCallback [602](#)
- servers, embedded HTTP [755](#)
- Service Activator patterns [111–112](#)
- ServiceCallConfigurationDefinition [745](#)
- ServiceCall EIP [772](#)

Service Call EIP

calling clustered services using [739–750](#)

configuring [745–747](#)

overview [740–741](#)

URI templating [747–748](#)

using Consul [748–750](#)

using Spring Boot Cloud [748–750](#)

using static registry service [741–745](#)

with failover [744–745](#), [749](#)

<serviceCall> element [743](#)

service level agreements. *See* SLAs (service level agreements)

ServiceMix [519](#), [529](#), [534](#)

service-oriented architecture. *See* SOA (service-oriented architecture)

Service Provider Interface. *See* SPI (Service Provider Interface)

services

calling [766–768](#), [770](#)

clustered, calling using Service Call EIP [739–750](#)

configuring [745–747](#)

overview [740–741](#)

URI templating [747–748](#)

using Consul [748–750](#)

using Spring Boot Cloud [748–750](#)

using static registry service [741–745](#)

with failover [744–745](#)

dealing with failures by calling in Kubernetes [789–792](#)

Kubernetes [781–783](#)

 changing port numbers [775–776](#)

 writing unit tests calling [794–795](#)

Servlet component [653](#)

SES (Simple Email Service) [234](#)

setException method [471](#)

setExecutorServiceManager method [577](#)

setFault(true) method [472](#)

setRecoveryInterval [164](#)

setRepeatInterval method [231](#)

shell completion [765](#)

Shipley, Grant [801](#)

shopping cart prototype [266–269](#)

short transactions [561](#)

showAll option [688, 812](#)

ShrinkWrap [399](#)

shrinkwrap-resolver-impl-maven dependency [366](#)

shutdown, graceful [642](#)

Shutdown activity [572](#)

shutdown command [662](#)

ShutdownRoute method [629, 644, 645](#)

ShutdownRunningTask option [629](#)

ShutdownStrategy interface [642](#)

shutting down Camel [641–645](#)

 considerations regarding [645](#)

 shutting down application [643–645](#)

side-car pattern [779](#)

file extension [830](#)

filename [829](#), [830](#)

bean [828](#)

bodyAs [825](#)

properties [825](#), [828](#)

automatic type coerce [830](#)

contains [830](#)

equals [830](#)

greater than [830](#)

in [830](#)

less than [830](#)

null safe operator [832](#)

range [832](#)

regex [830](#), [832](#)

syntax [830](#)

with built-in functions [832](#)

date [830](#)

header [825](#)

Simple

expression [823](#), [834](#)

invoke method on message body [172](#)

OGNL [833](#)

operator [824](#)

predicate [823](#), [824](#), [834](#)

SimpleBuilder [834](#)

splitting message body [94](#)
using as expression [83](#)
using from Processor [833](#)
using with [823](#)
variables binding to Exchange [824](#)

SimpleDataFormat [202](#)

Simple expression language [823–834](#)
built-in file variables [828–830](#)
built-in functions [825–828](#)
built-in operators [830–832](#)
built-in variables [824–825](#)
OGNL feature [832–833](#)
overview [823–824](#)
syntax [824](#)
using Simple from custom Java code [833](#)

SimpleLoadBalancerSupport class [191](#)

Simple Queue Service. *See* SQS (Simple Queue Service)

SimpleRegistry [114](#), [115–116](#)

<simple> tag [172](#)

simulating
errors [378–390](#)
adding interceptors to existing routes with adviceWith method [382–383](#)
amending routes using weave with adviceWith method [387–390](#)
manipulating routes with adviceWith method [383–387](#)
with interceptors [381–382](#)

with mocks [380–381](#)
with processors [378–380](#)
real components with mock components [376–378](#)
SJMS (Simple, Standard, and Spring-less) [206](#)
skipBindingOnErrorCode option [436](#)
skipSendToOriginalEndpoint method [382](#)
SLAs (service level agreements) [28, 279, 516, 565, 671](#)
sleep method [12](#)
slip method [181](#)
smart endpoints [244](#)
smart endpoints and dumb pipes [244](#)
SMTP (Simple Mail Transfer Protocol) [234–235](#)
SNMP (Simple Network Management Protocol) [672](#)
SNS (Simple Notification Service) [244](#)
Snyder, Bruce [32, 527](#)
SOA (service-oriented architecture) [244, 599](#)
SOAPFaultException [604, 607](#)
SoapUI [544, 651](#)
source-first approach [445](#)
specifying, destinations with URIs [33](#)
SPI (Service Provider Interface) [113](#)
SplitDefinition [389](#)
splitDepartments method [171, 172](#)
<split> tag [173](#)
Splitter EIPs
aggregate [169–174](#)

aggregating split messages [174–175](#)
 streaming mode [173](#)
 using stream [172](#)
combine message [174](#)
errors during splitting [176–178](#)
 handling exceptions using AggregationStrategy [176–178](#)
 using stopOnException [176](#)
Exchange properties [170](#)
Expression [169](#)
 by AggregationStrategy [177](#)
 by stopping [176](#)
 in AggregationStrategy [176](#)
iterate [169](#)
 overview [147, 169–170](#)
 split complete [170](#)
 split index [170](#)
 split size [170](#)
 splitting message body [92](#)
 splitting messages [172–173](#)
 tokenizer [173](#)
 using [168](#)
 using beans for splitting [170–172](#)
 using java.util.Scanner [173](#)
splitting
 errors during [176–178](#)
 handling exceptions using AggregationStrategy [176–178](#)

using stopOnException [176](#)
messages [172–173](#)
using beans for [170–172](#)

Spring
 bean wiring example [40](#)
 ClassPathXmlApplicationContext class [41](#)
 context listener [649](#)
 loading CamelContext in Spring XML [42](#)
 property placeholder [50](#)
 separate wiring into XML files [48](#)
 wiring [41](#)

SpringBootApplication.java file [264](#)
SpringBootApplication class [301](#)
Spring Boot Cloud, Service Call EIP with [748–750](#)
Spring Boot framework
 adding Camel to existing REST endpoint [258](#)
 applications [257–258](#)
 as microservice [256–262](#)
 configuring with Rest DSL [441](#)
 Hystrix with [293–295](#), [298–301](#)
 monitoring Camel with [262](#)
 overview [256–257](#)
 property placeholders with [261](#)
 routes with [259–260](#)
 running without embedded HTTP servers [755](#)
 starter components with [259](#)

testing with [356–358](#)
XML DSL with [260–261](#)
Spring Boot pod [781, 782, 784](#)
SpringCamelContext class [42](#)
SpringCamelTestSupport [350](#)
Spring framework
 containers, running Camel in [46–49](#)
 configuring components [47–48](#)
 configuring endpoints [47–48](#)
 finding route builders [46–47](#)
 importing configuration [48–49](#)
 importing routes [48–49](#)
 setting advanced configuration options [49](#)
 testing with [349–368](#)
 transaction support [522–523](#)
 using property placeholders in [52–53](#)
 using to create applications from beans [40–43](#)
Spring Java Config, testing with [352–355](#)
SpringJUnit4ClassRunner [352](#)
Spring Security, configuring [619–621](#)
SpringSecurityAuthorizationPolicy class [619, 620](#)
Spring XML (extensible markup language)
 routes with [255](#)
 testing with [350–352](#)
sql.properties file [517](#)
SQLException [470](#)

SQL INSERT statement [517](#)
SQS (Simple Queue Service) [244](#)
SSL (Secure Sockets Layer) [203](#)
SSLContextParameters [615–617](#)
Stack Overflow website, Camel community at [836](#)
staged event-driven architecture. *See* SEDA (staged event-driven architecture)
starting Camel [626–635](#)
 disabling AutoStartup [634–635](#)
 options for [629–631](#)
 configuring HandleFault [631](#)
 configuring StreamCaching [630](#)
 ordering routes [631–634](#)
 how StartupOrder works [633–634](#)
 using StartupOrder to control [631–633](#)
 process for [626–628](#)
 preparing CamelContext in Spring container [627](#)
 starting CamelContext [627–628](#)
start method [626, 628](#)
startRoute method [635, 636](#)
startsWith method [373](#)
StartupOrder
 function of [633–634](#)
 using to control ordering of routes [631–633](#)
State attribute [694](#)
stateful mode [718](#)

stateless mode [718](#)
states. *See* declarative states
static registry service, Service Call EIP with [741–745](#)
Sticky strategy [186](#)
stop() method [60](#), [642](#)
stopOnException method [64](#), [176](#)
stop operation [694](#), [696](#)
stopRoute method [635](#), [645](#)
StopRouteProcessor class [636](#)
Strachan, James [801](#)
strategyRef attribute [87](#), [150](#)
StreamCaching, configuring [630](#)
streaming attribute [173](#)
streaming mode [564](#)
stream magic, Java 8 [293](#)
StreamMessage [211](#)
streams, Hystrix [301](#)
Stub component [198](#)
stubs, replacing JMS with [371](#)
submessages [147](#)
substitutions [331](#)
super.configure method [491](#), [492](#)
super bundles [660](#)
Swagger2Feature [448](#)
Swagger framework
 adding to CXF applications [448–451](#)

for API documentation [444–465](#)
 configuring [459–461](#)
 documenting error codes [454–459](#)
 documenting input [454–459](#)
 documenting output [454–459](#)
 documenting Rest DSL services [452–453](#)
 using CORS [461–465](#)

JAX-RS with [446–451](#)
 overview [445–446](#)

Rest DSL with [451–452](#)
 web console [461–465](#)

symmetric cryptography [612](#)

sync=true option [87](#)

SynchronizationAdapter class [546](#)
 synchronization callbacks [546–547](#)
 synchronous messaging [227](#)
 synchronous receiver [24](#)
 synchronous throttling [578](#)
 syntax, in Simple [824](#)

System.out.println [688, 700](#)

System.out stream [201](#)

T

tag[failover] [190](#)
tag[loadBalance] [192](#)
tag[split] [176](#)

tar.gz distribution [10](#)
task queue [573](#)
TCP (Transmission Control Protocol) [213](#)
TCP Socket probe [786](#)
templating
 transforming with [99](#)
 URIs with Service Call EIP [747–748](#)
test attribute [61](#)
testAuditLogFail method [541](#)
testAuditLog method [541](#)
test classes, writing with deployment units [398–399](#)
Test component [198](#)
testing
 Camel Test Kit [345–349](#)
 Camel JUnit extensions [346](#)
 testing Java Camel routes with camel-test [346–349](#)
 unit testing existing RouteBuilder classes [349](#)
 integration [390–395](#)
 inspecting queues [392–393](#)
 NotifyBuilder and [393–395](#)
 sending order messages using clients [391](#)
 waiting for applications to process messages [392](#)
 microservices on Kubernetes [792–795](#)
 running Arquillian Cube tests on Kubernetes [793–794](#)
 setting up Arquillian Cube [792](#)
 writing unit tests calling Kubernetes services [794–795](#)

writing unit tests using Arquillian Cube [793](#)
mock components for [368–378](#)
 overview [369](#)
 to simulate real components [376–378](#)
 unit testing with [369–371](#)
 using expressions with [372–376](#)
 verifying arrival of correct messages [371–372](#)
simulating errors [378–390](#)
 adding interceptors to existing routes with adviceWith method [382–383](#)
 amending routes using weave with adviceWith method [387–390](#)
 manipulating routes with adviceWith method [383–387](#)
 with interceptors [381–382](#)
 with mocks [380–381](#)
 with processors [378–380](#)
 transactions [526–529](#)
 using third-party testing libraries [395–405](#)
 Arquillian [395–399–405](#)
 Awaitility [404](#)
 Byteman [404](#)
 Citrus integration testing [404](#)
 Gatling [405](#)
 JMC [405](#)
 JMH [405](#)
 Pax Exam [399–404](#)
 Wireshark [405](#)

YourKit [405](#)
with CDI [349–368](#)
with only one RouteBuilder class enabled [357–358](#)
with OSGi [349–368](#)
with OSGi Blueprint XML [358–361](#)
with Spring [349–368](#)
with Spring Boot [356–358](#)
with Spring Java Config [352–355](#)
with Spring XML [350–352](#)
with WildFly [364–368](#)
 installing WildFly [367–368](#)
 testing WildFly using Arquillian [366–367](#)
with WildFly Swarm [363–364](#)
testing externalize dynamic parts [50](#)
Test Kit [9](#)
testMiranda method [377](#)
testMoveFile method [348, 351](#)
testNoConnectionToDatabase [526](#)
testOrderActiveMQ method [488](#)
testWithCamel method [541](#)
testWithDonkey method [541](#)
TextMessage [211](#)
Thread.sleep call [25](#)
threadFactory option [572](#)
thread pool [64, 566, 571](#)
ThreadPoolExecutor class [571](#)

ThreadPoolProfileBuilder class [574](#)
<threadPoolProfile> tag [574](#), [575](#)
thread pools
 creating custom [567–568](#), [575–576](#)
 in Java DSL [575](#)
 in XML DSL [576](#)
 in Java [571–573](#)
 managing [572](#)
 profiles [573–575](#)
 configuring custom [574–575](#)
 configuring default [574](#)

Threads EIP, parallel processing [578–581](#)
<threads> tag [576](#)

Throttler EIP [578](#), [703](#)

Timeout option [227](#)

time-outs

 overview [279](#)
 with Netflix Hystrix [289–290–291](#)

Timer component [198](#)

TimeUnit type [572](#)

TLS (Transport Layer Security) [203](#), [596](#)

toD, using as dynamic to [143–145](#)

tokenizer [173](#)

to keyword [33](#)

tooling [803–822](#)

 Camel Catalog [813–816](#)

Camel community [837](#)
Camel editors [804–813](#)

- Apache Camel IDEA plugins [808–810](#)
- JBoss Fuse Tooling [804–808](#)
- validation using Maven [811–813](#)

hawtio [816–822](#)

- best practices [821–822](#)
- debugging routes using [818–821](#)
- functionality of [817–818](#)

topic [31](#)
Topic strategy [186](#)
toString method [554](#)
to style, using instead of beans [111](#)
<to> tag [53](#)
TRACE logging level [685](#)
Tracer feature [689–691](#)
tracers, integrated [806–808](#)
tracing [316](#)
Tracing option [629, 630](#)
tracking application activity [684–694](#)

- using core logs [685](#)
- using custom logging [685–689](#)
- using log files [685](#)
- using notifications [691–694](#)
- using Tracer [689–691](#)

transacted acknowledge mode [521](#)

<transacted/> tag [525](#)

transaction, JmsTransactionManager [524](#)

Transactional Client EIPs [529–544](#)

- global transactions [531–534](#)
- local transactions [530–531](#)
- returning custom responses when transactions fail [542–544](#)
 - catching exceptions [542–543](#)
 - constructing reply messages [543–544](#)
 - handling exceptions [542–543](#)
 - rolling back transactions [543](#)
- transaction propagations [539–541](#)
- transaction redeliveries [536–539](#)
 - configuring broker-level redelivery with ActiveMQ [538](#)
 - configuring consumer-level redelivery with ActiveMQ [537](#)
 - starting from databases [538–539](#)
- transactions starting from database resources [534–536](#)

TransactionErrorHandler [474, 480, 526](#)

TransactionManager [165](#)

transactions [514–561](#)

- adding [524–526](#)
- advantages of using [515–521](#)
 - lost messages [519–521](#)
 - setting up databases [518–519](#)
 - setting up JMS brokers [518–519](#)
- global [531–534](#)
- local [530–531](#)

overview [521–529](#)

propagations [539–541](#)

redelivery [536–539](#)

- configuring broker-level redelivery with ActiveMQ [538](#)
- configuring consumer-level redelivery with ActiveMQ [537](#)
- starting from databases [538–539](#)

returning custom responses when failing [542–544](#)

- catching exceptions [542–543](#)
- constructing reply messages [543–544](#)
- handling exceptions [542–543](#)
- rolling back transactions [543](#)

starting from database resources [534–536](#)

testing [526–529](#)

Transactional Client EIPs [529–544](#)

- unsupported [544–551](#)

transform DSL method [215](#)

transforming

- data [75–105](#)
 - overview [76–77](#)
 - using <transform> in XML DSL [83](#)
 - using beans [80–81](#)
 - using EIPs [77–87](#)
 - using Java [77–87](#)
 - using processors [78–80](#)
 - using transform method in Java DSL [81–](#)
 - with type converters [99–105](#)

with data formats [91–99](#)

- configuring data formats [97–99](#)
- providing data formats [92](#)
- using Bindy data formats [94–97](#)
- using CSV data formats [92–94](#)
- using JSON data formats [97](#)
- with templates [99](#)
- with XStream [89](#)
- XML [87–91](#)
 - with object marshaling [88–91](#)
 - with XSLTs [87–88](#)
- transform method, in Java DSL [81–83](#)
- <transform> tag, in XML DSL [83](#)
- transitive dependencies [311](#)

Transmission Control Protocol. *See* TCP (Transmission Control Protocol)

Transport Layer Security. *See* TLS (Transport Layer Security)

transport security [614–617](#)

trigger.repeatCount property [231](#)

trigger.repeatInterval option [231](#)

troubleshooting, method-selection [125–127](#)

trustManagers [615](#)

trust store [609](#), [611](#), [615](#), [616](#)

type converter

- data type transformation using [104](#)
- overview [76](#)

TypeConverter parameter [130](#)
TypeConverterRegistry [101](#)
type converters [99–105](#)
 adding to Camel-core [104–105](#)
 automatic [9](#)
 encoding [103](#)
 loading into registry [101–102](#)
 overview [101–102](#)
 using [102–103](#)
 writing [103–104–105](#)
type matching, selecting methods with [127–128](#)
type option, JSON schema [816](#)
types, built-in [130](#)
type-safe endpoint editors [806](#)
TypeScript [821](#)

U

UDP (User Datagram Protocol) [213](#)
Unique thread names activity [572](#)
UnitOfWork [545–546](#)
unit option [572](#)
unit testing
 calling Kubernetes services [794–795](#)
 using Arquillian Cube [793](#)
unit tests
 debugging [317–318](#)

improving [347–349](#)
of existing RouteBuilder classes [349](#)
setting up [335–336](#)
with @RunWith [351–352](#), [354–355](#)
with mock components [369–371](#)

unmarshaling [87](#)

unmarshal method [91](#), [338](#), [340](#)

unsupported transactions

- compensating for [544–551](#)
- onCompletion [548–551](#)
- synchronization callbacks [546–547](#)
- UnitOfWork [545–546](#)

UpdateInventoryInput object [632](#)

Uptime attribute [674](#)

UriEndpointComponent class [320](#)

UriPath [323](#)

URIs (uniform resource identifiers)

- referencing registry beans in endpoints [54](#)
- templating with Service Call EIP [747–748](#)
- using property placeholders in endpoints [50–53](#)
- using raw values in endpoints [54](#)
- using to specify destinations [33](#)

URI templating [739](#)

--url parameter [763](#), [767](#)

UseExponentialBackOff option [483](#)

useFixedDelay option [230](#)

useHeadersAsParameters option [220](#)
useOriginalMessage option [478](#)
useRecovery option [166](#)
<UsernameToken> element [600](#)
Using AggregateController condition [153](#)

V

validate method [474, 475, 478](#)
ValidateProcessor [504](#)
validating Camel using Maven [811–813](#)
ValidationException [503, 504](#)
ValidatorBean [210](#)
Validator component [198](#)
values, comma-separated [66](#)
variables, in Simple [824–830](#)
Velocity [99](#)
Verbose attribute [710](#)
verify endpoint [611](#)
version element [14](#)
Vert.X [245](#)
videos, in Camel community [837](#)
VisualVM-MBeans plugin [676](#)
Vlissides, John [78](#)
VM (virtual machine) [766](#)
VM components [226–230](#)
VM protocol [519](#)

VM transport [685](#)

vm transport connector [32](#)

W

WADL (Web Application Description Language) [445](#)

Walls, Craig [39](#)

WAR agents, Jolokia [681–684](#)

WARN [473, 506, 526, 560](#)

weaveAddLast [388](#)

weaveById [387](#)

weaveByToString [390](#)

weaveByUri [390](#)

weaveByType [389](#)

weaving

 and adviceWith method [387–390](#)

 selecting nodes with [389–390](#)

 without using IDs [388–389](#)

web applications, embedding Camel in [648–654](#)

 deploying to Apache Tomcat [650–652](#)

 using Apache Tomcat for HTTP inbound endpoints
[652–654](#)

 webapps directory, Apache Tomcat [651](#)

 web consoles [816–822](#)

 best practices [821–822](#)

 connecting to existing running JVM [820–821](#)

 debugging routes using [818–821](#)

functionality of [817–818](#)

Swagger UI, embedding [463–465](#)

WEB-INF directory [649](#)

web service security, authentication [600–604](#)

- client-side WS-Security configuration [603–604](#)
- server-side WS-Security processing [601–602](#)
- using JAAS [604–607](#)

WebSphere MQ [47](#)

wereSentTo method [394](#)

whenAnyDoneMatches method [393, 394](#)

whenAnyExchangeReceived method [376, 377, 381](#)

whenBodiesDone method [394](#)

whenCompleted method [394](#)

whenDone method [394](#)

whenExchangeReceived method [376](#)

whenFailed method [394](#)

when method [56](#)

whereToGo method [182](#)

wildcards [382](#)

WildFly

- installing [367–368](#)
- running Camel in [654–658](#)
- testing using Arquillian [366–367](#)
- testing with [364–368](#)

wildfly-arquillian-container-managed dependency [366](#)

WildFly-Camel Subsystem [655–658](#)

WildFly Swarm

as microservice [252–256](#)
fragments versus Camel components [254–255](#)
Hystrix with [296–297](#)
monitoring Camel with [255–256](#)
testing with [363–364](#)
with C for microservices [256](#)

WildFlySwarmCamelTest class [363](#)
Wireshark, testing applications with [405](#)
Wire Tap EIP, parallel processing [583–586](#)
wireTap method [69–71](#)
wire taps, using for logging [686–687](#)
wiring [41](#)
wmq:myWebSphereQueue [47](#)
Woolf, Bobby [7, 529](#)
workQueue option [572](#)
writing
 custom data formats [339–342](#)
 files with File components [200–203](#)
 integration test classes [402–403](#)
 test classes with deployment units [398–399](#)
 type converters [103–105](#)

WSCallbackHandler [605](#)
WS-Choreography (Web Service Choreography) [244](#)
WSDL (Web Services Description Language) [445, 599](#)
WSPasswordCallback [604](#)

WSS4JInInterceptor [605](#)

WSS4JOutInterceptor [603](#)

WS-Security

client-side configuration [603–604](#)

server-side processing [601–602](#)

to/from POJOs using camel-jaxb [419](#)

with Rest DSL [439–441](#)

X

XML (extensible markup language)

data formats, with Rest DSL [434–443](#)

defining routes in [39–49](#)

creating applications from beans using Spring [40–43](#)

running Camel in Spring containers [46–49](#)

DSLs [43–45](#)

adding processors [44–45](#)

choosing [45](#)

invoking beans defined in [108–109](#)

using multiple routes [45](#)

using property placeholders in [51–52](#)

with Spring Boot [260–261](#)

JSON instead of [413–414](#)

marshaling [87](#)

serializing objects to and from [90](#)

transforming [87–91](#)

with object marshaling [88–91](#)

with XSLTs [87–88](#)

xmlDataFormat option [436](#)

XML DSL

- configuring JMX Agent from [679–680](#)
- creating custom thread pools [576](#)

XML elements [42](#)

xml mode [434](#)

xmlOrders queue [57, 61, 208, 209, 229, 230](#)

XML programming language, using <transform> to transform data [83](#)

XPath [56, 62, 220](#)

XQuery [56](#)

XSLTs (XSL Transformations)

- overview [198](#)
- transforming XML with [87–88](#)

XStream library, transforming with [89](#)

Y

YAML format [767](#)

YourKit tool, testing applications with [405](#)

Z

Zipkin [691, 801](#)

Zookeeper service, clustered active/pассив mode [724–726](#)