

Camel IN ACTION



SECOND EDITION

Claus Ibsen
Jonathan Anstey



MEAP



MANNING



**MEAP Edition
Manning Early Access Program
Camel in Action
Second Edition
Version 12**

Copyright 2017 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Camel in Action 2nd edition*. We are thrilled to be back after 5 years and writing about Apache Camel again. Readers of the 1st edition will get the same level of dedication and hard work we put into the first book. We have taken all the best chapters from the first edition and updated them with all the latest functionality that Apache Camel offers together with all the excellent feedback we have received over the years about the book. We have also not been afraid of dropping content from those chapters which are less attractive today to add in more content.

It's early 2017 and we are back on track.

Claus handed in a staggering 80 page chapter on microservices. We guess microservices aren't so micro after all. If they were, then the chapter would only be a few pages.

However, he already has some thoughts on breaking up the chapter and moving the reactive/vert.x coverage to a new independent chapter. Fingers crossed this goes forward (one more chapter for you) and you will have a lighter microservices chapter in the future.

Jonathan handed in chapter 15, which covers how to run Camel in many different containers, servers and what have you. The chapter also discusses the Camel startup and shutdown procedure, which surprisingly is harder to do reliably than you may think. Don't miss out on this as you can learn many tips on how to ensure your production team is happy about Camel applications.

We have also updated Chapter 12,13,16, and 19 based off reviewer feedback

We love your feedback so please post to the Manning author forum. Or shoot us an email at: claus.ibsen@gmail.com or janstey@gmail.com.

Or if you can keep it short then a tweet to @davsclaus or @jon_anstey is fine as well.

Regards

—Claus Ibsen and Jonathan Anstey

PS: We are considering a bonus chapter about using Camel with IoT. If you see this as valuable information then let us know.

brief contents

PART 1: FIRST STEPS

- 1 Meeting Camel*
- 2 Routing with Camel*

PART 2: CORE CAMEL

- 3 Transforming data with Camel*
- 4 Using beans with Camel*
- 5 Enterprise Integration Patterns*
- 6 Using components*

PART 3: DEVELOPING AND TESTING

- 7 Microservices*
- 8 Developing Camel projects*
- 9 Testing*

PART 4: GOING FURTHER WITH CAMEL

- 10 REST and web services*
- 11 Error handling*
- 12 Transactions and Idempotency*
- 13 Parallel processing*
- 14 Securing Camel*

PART 5: RUNNING AND MANAGING CAMEL

- 15 Running and Deploying Camel*
- 16 Management and monitoring*
- 17 Clustering Camel*

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

PART 6: OUT IN THE WILD

18 Camel in the cloud

19 Camel Tooling

APPENDIXES:

A Setting up your IDE

B The Simple Language

C Using alternative DSLs

D Contributing to Apache Camel

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 one of Camel's most important features: message routing. Camel has two main ways of defining routing rules: the Java-based domain-specific language (DSL) and the Spring XML configuration format. In addition to these route-creation techniques, we'll show you how to design and implement solutions to enterprise 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 very 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 healthcare 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 an integrator, 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 will start off by introducing Camel and highlighting some of its core features. We'll then take a look at the Camel distribution and explain how you can 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.

The Apache Camel project was named Camel simply because the name is short and easy to remember. Rumor has it the name may be inspired by the fact that one of the founders once smoked Camel cigarettes. 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.

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 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 about 200 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 very 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 enterprise 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, monitoring, orchestration, 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 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, do not fear. Camel is equally at home in tiny deployments such as microservices or internet of things (IoT) gateways.

To understand what Camel is, it helps to look at its main features. So let's take a look at them.

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, instead of using an existing framework. We'll explore the rich set of Camel features throughout the book, but these are the main ideas behind Camel:

| | |
|----------------------------------|--|
| • Routing and mediation engine | • Enterprise integration patterns (EIPs) |
| • Domain-specific language (DSL) | • Extensive component library |
| • Payload-agnostic router | • Modular and pluggable architecture |
| • POJO model | • Easy configuration |
| • Automatic type converters | • Lightweight core |
| • 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 will selectively move 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 about 200 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.



Figure 1.2 Connect to just about anything! Camel supports about 200 different transports, APIs, and data formats.

ENTERPRISE INTEGRATION PATTERNS (EIPS)

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*, a must-read for any integration professional (<http://www.enterpriseintegrationpatterns.com>). If you haven't read it, we encourage you to do so. At the very least, it will 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. The difference between using a pattern language and describing the problem at hand is

similar to using spoken language rather than sign language. If you've ever visited a foreign country, you've probably experienced the difference.

Camel is heavily based on EIPs. Although EIPs describe integration problems and solutions and also 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 (DSL)

At its inception, Camel's domain-specific language (DSL) was a major contribution to the integration space. Since then many other integration frameworks have followed suit and now feature DSLs in Java, XML or custom languages. Camel is unique because it offers multiple DSLs in regular programming languages such as Java, Scala, Groovy, and it also allows routing rules to be specified in XML.

- The purpose of the DSL is to allow the developer to focus on the integration problem rather than on the tool—the programming language. Although Camel is written mostly in Java, it does support mixing multiple programming languages. Each language has its own strengths, and you may want to use different languages for different tasks. You have the freedom to build a solution your own way with as few constraints as possible.
- Here are some examples of the DSL using different languages 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>
```

- Scala DSL

```
from "file:data/inbox" -> "jms:queue:order"
```

- Groovy DSL

```
from "file:data/inbox" to "jms:queue:order"
```

These examples are real code, and they show how easily you can route files from a folder to a 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.

```

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 like Java, so you can use existing tooling support.

Here you can see how the Eclipse IDE's autocomplete feature can give us 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 XML payloads. This freedom means that 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

Beans (or POJOs) are considered first-class citizens in Camel, and Camel strives to let you use beans anywhere and anytime in your integration projects. This means that 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 file consumer to scan recursively in a subfolder and include only a .txt file, 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 find a 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 leverage 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

Camel's core can be considered pretty lightweight, with the total library coming in at about 3.3 MB and only having 1.4 MB of 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 Kubernetes and Cloud Foundry. Camel was designed not to be a server or ESB but instead to be embedded in whatever platform you choose. You just need Java.

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 16,000 unit tests. The Test Kit contains test-specific components that, for example, can help you mock real endpoints. It also contains setup 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 over 9 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 promptly. For more information on Camel's community, see appendix D.

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

1.2 Getting started

In this section, we'll show you how to get your hands on a Camel distribution, explain what's inside, and then 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 also the downloads for the latest release.

- For the purposes of this book, we'll be using Camel 2.17.0. To get this version, click on the Camel 2.17.0 Release link and 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 Mac OS X/Linux users. When you've downloaded one of the distributions, extract it to a location on your hard drive.
- Open up 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@bender apache-camel-2.17.0]$ ls
doc examples lib LICENSE.txt NOTICE.txt README.txt
```

As you can see, the distribution is pretty 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 wiki at the time of release. As such, it's a decent reference for those not able to access the Camel website (or if you misplaced your copy of Camel in Action).
- examples—Includes 64 Camel examples. You'll see an example shortly.
- lib—Contains all Camel libraries and third-party dependencies needed for the core of Camel to run. You'll see later in the chapter how Maven can be used to easily grab 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 what Camel is and a list of helpful links to get new users up and running fast.

Now let's try out one of the Camel examples.

1.2.2 Your first Camel ride

So far, we've shown you how to get a Camel distribution and we've explored 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. The examples in the book's source all 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 path, for example.

You can get the book's source code from either the book's website, at <https://www.manning.com/books/camel-in-action-second-edition> or from the GitHub project that's hosting the source: <https://github.com/camelinaction/camelinaction2>.

The first example we'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.

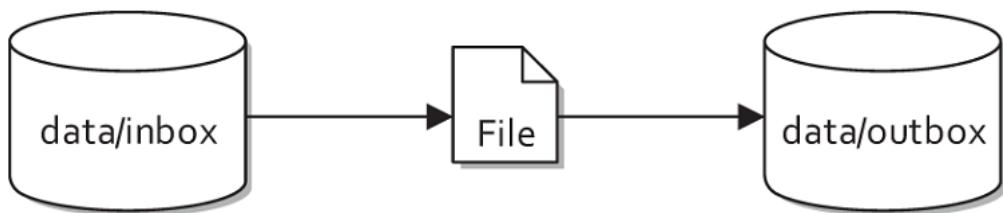


Figure 1.4 Files are routed from the `data/inbox` directory to the `data/outbox` directory.

It looks pretty 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();
    }
}
}

```

The `FileCopier` example in listing 1.1 is a pretty simple use case, but it still results in 34 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 wanted to poll the `data/inbox` directory for new files, you'd need to set up a timer and also 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 like 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");          ②
            }
        });
        context.start();
        Thread.sleep(10000);
    }
}

```

```

        context.stop();
    }
}

```

① Routes files from inbox to outbox

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 really 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 have 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 one line of Java code ①.

To run this example, you'll need to download and install Apache Maven from the Maven site at <http://maven.apache.org/download.html>. If you prefer to stay within your IDE, you can find setup instructions in Appendix A. Once 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 3.3.3 during the development of the book. Different versions of Maven may not work or appear exactly as we've shown.

The POM is shown here.

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.camelinaaction</groupId>
    <artifactId>chapter1</artifactId>
    <version>2.0.0</version>
  </parent>

```

①
①
①

```

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

<dependencies>
  <dependency>
    <groupId>org.apache.camel</groupId>          ②
    <artifactId>camel-core</artifactId>            ②
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>                  ③
    <artifactId>slf4j-log4j12</artifactId>        ③
  </dependency>
</dependencies>
</project>

```

- ① Parent POM
- ② Camel's core library
- ③ Logging support

Maven itself is a complex topic, and we won't go into great detail here. We'll give you enough information to be productive with the examples in this book. For an in-depth look at Maven, we recommend reading *Maven by Example* and *Maven: The Complete Reference*, both of which are freely available from <http://books.sonatype.com>. We'll also discuss using Maven to develop Camel applications in chapter 8, 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, there are also some settings 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 will resolve to 2.17.0. The type of dependency was not specified, so the JAR file type will be assumed. The search path will be `org/apache/camel/camel-core/2.17.0/camel-core-2.17.0.jar`.
2. Because listing 1.3 defined no special places for Maven to look for the Camel dependencies, it will look in Maven's central repository, located at <http://repo1.maven.org/maven2>.
3. Combining the search path and the repository URL, Maven will try to download <http://repo1.maven.org/maven2/org/apache/camel/camel-core/2.17.0/camel-core-2.17.0.jar>.
4. This JAR will be saved to Maven's local download cache, which is typically located in the home directory under `.m2/repository`. This would be `~/.m2/repository` on Linux/Mac OS X 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 will be 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 -Dexec.mainClass=camelinaction.FileCopierWithCamel
```

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 In order to run any of the examples in this book you'll need an Internet connection. A broadband speed connection is preferable because Apache Maven will download many JAR dependencies of the examples, some of which are large. The whole set of examples will download about 140 MB of libraries.

TODO recheck this when book is complete!

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 has just been made. Congratulations, you've just run your first Camel example! It was a simple example, 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 some 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 service-oriented architecture (SOA) and EIP concepts and retain their names and semantics. We'll start with Camel's message model.

1.3 Camel's message model

In Camel, there are two abstractions for modeling messages, both of which we'll 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 Message to understand how data is modeled and carried in Camel. Then we'll look at 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.

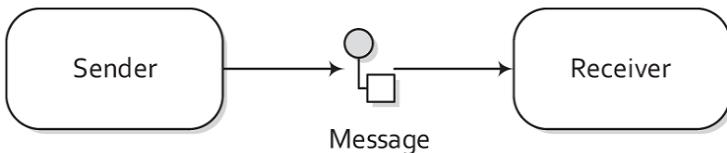


Figure 1.5 Messages are entities used to send data from one system to another.

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

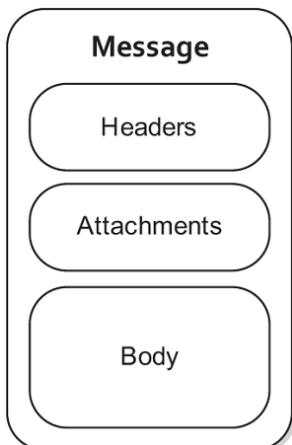


Figure 1.6 A message can contain headers, attachments, and a body.

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 UID 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`. This means that Camel imposes no constraints on the type of the headers. There are also no constraints on the size of headers or on how many headers are 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`. That means that a message can store any kind of content and also any size. It also means that 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 a number of mechanisms to transform the data into an acceptable format, and in many cases the conversion happens automatically with type converters, behind the scenes. We cover message transformation fully in chapter 3.

FAULT FLAG

Messages also have a fault flag. Some protocols and specifications, such as WSDL and JBI, 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:

- `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, where a client requests to retrieve a web page, waiting for the reply from the server.

Figure 1.7 illustrates the contents of an exchange in Camel.

Let's look at the elements of figure 1.7 in more detail:

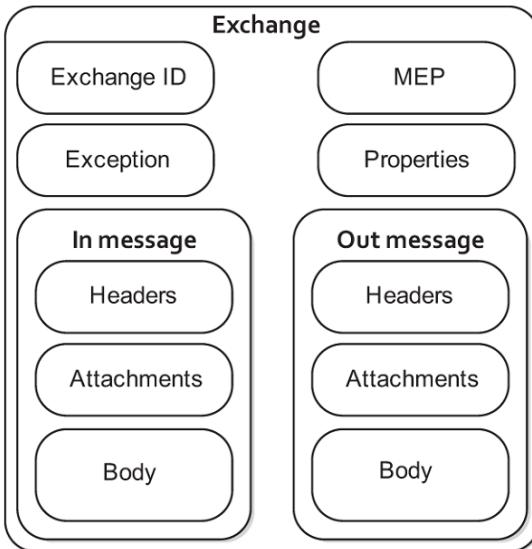


Figure 1.7 A Camel exchange has an ID, MEP, exception, and properties. It also has an *in* message to store the incoming message and an *out* message to store the result.

- **Exchange ID**—A unique ID that identifies the exchange. Camel will generate a default unique ID, if you don't explicitly set one.
- **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 will add 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 only exists if the MEP is `InOut`. The *out* message contains the reply message.

The Exchange is the same for the entire life cycle 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 wanted 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

Let's now turn our attention to Camel's architecture. We'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 we'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.

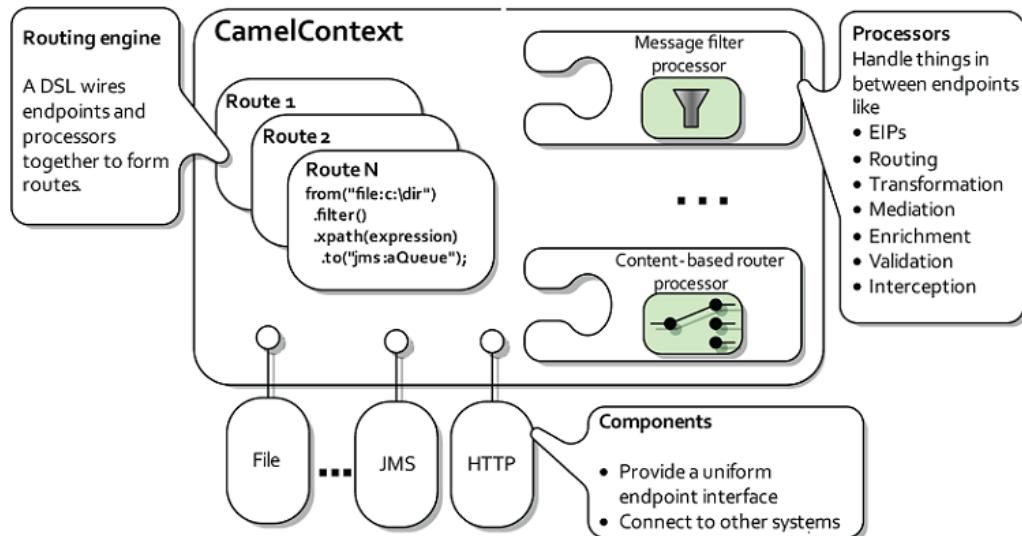


Figure 1.8 At a high level, Camel is composed of processors, components, and routes. All of these are contained within the `CamelContext`.

The routing engine uses routes as specifications for where messages are routed. Routes are defined using one of Camel's domain-specific languages (DSLs). Processors are used to transform and manipulate messages during routing and also to implement all the EIP patterns, which have corresponding keywords in the DSL languages. 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 revealed many new concepts, so let's take some time to go over them one by one. We'll start with the `CamelContext`, which is Camel's runtime.

CAMELCONTEXT

You may have guessed that the `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 the `CamelContext` keeps together.

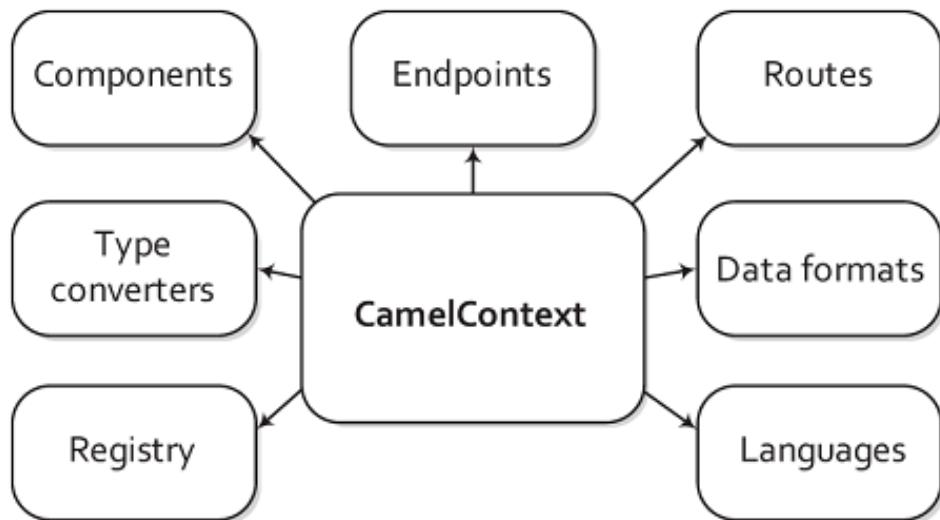


Figure 1.9 The `CamelContext` provides access to many useful services, the most notable being components, type converters, a registry, endpoints, routes, data formats, and languages.

As you can see from figure 1.9, there are a lot of services for the `CamelContext` to keep track of. These are described in table 1.1.

The details of each of these services will be discussed throughout the book. Let's now take a look at routes and Camel's routing engine.

Table 1.1 The services that the 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. In chapter 6 we'll discuss components in more detail. |
| Endpoints | Contains the endpoints that have been created. |
| Routes | Contains the routes that have been added. We'll cover routes in chapter 2. |
| 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. By default, this will be a JNDI registry. If you're using Camel from Spring, this will be the Spring ApplicationContext. If you are using Camel with CDI, this will be backed by a CDI BeanManager. It can also be an OSGi registry if you use Camel in an OSGi container. We'll cover registries in chapter 4. |
| Languages | Contains the loaded languages. Camel allows you to use many different languages to create expressions. You'll get a glimpse of the XPath language in action when we cover the DSL. A complete reference to Camel's own Simple expression language is available in appendix A. |

ROUTING ENGINE

Camel's routing engine is what actually 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

- 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
- Allow for clients of servers to be stubbed out (using mocks) for testing purposes
- 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)

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 is 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 3 separate routes. So, it behaves similar to 3 separate routes like:

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

DOMAIN-SPECIFIC LANGUAGE (DSL)

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 a test order or not. If a message passes the test, it's forwarded to the JMS endpoint. Messages failing the filter test will be dropped.

Camel provides multiple DSL languages, so you could define the same route 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 actually composed of a graph of processors. Let's take a moment to see what a processor really 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.

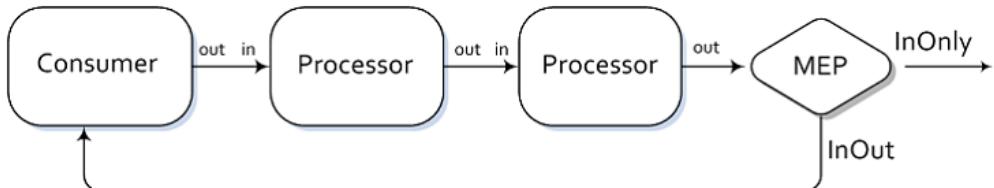


Figure 1.10 Flow of an exchange through a route.

Figure 1.10 shows the flow between processors visually. 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 do not actually set an out message so in this case the in message is reused. At the end of a route the MEP of the exchange will determine if a reply needs to be sent back to the caller of the route. If the MEP is InOnly then no reply will be sent back. If it is InOut, then Camel will take the out message from the last step and return it. In this final step the out message **must** be set for a reply to be sent back – the in message will not be reused here.

So how do exchanges get in or out of this processor graph? To find out, we’ll need to look at both components and endpoints.

COMPONENT

Components are the main extension point in Camel. To date, there are over 200 components in the Camel ecosystem that range in function from data transports, to DSLs, data formats, and so on. You can even create your own components for Camel—we’ll 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, a `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.

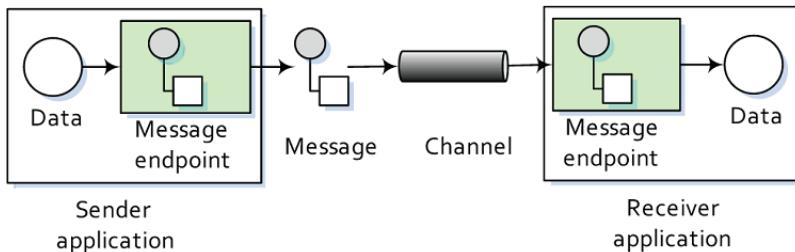


Figure 1.11 An endpoint acts as a neutral interface allowing systems to integrate.

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

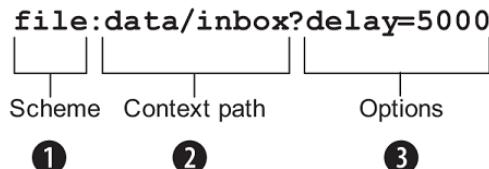


Figure 1.12 Endpoint URIs are divided into three parts: a scheme, a context path, and options.

The scheme ① denotes which Camel component handles that type of endpoint. In this case, the scheme of `file` selects the `FileComponent`. The `FileComponent` then works as a factory creating the `FileEndpoint` based on the remaining parts of the URI. The context path `data/inbox` ② tells the `FileComponent` that the starting folder is `data/inbox`. The option, `delay=5000` ③ 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.

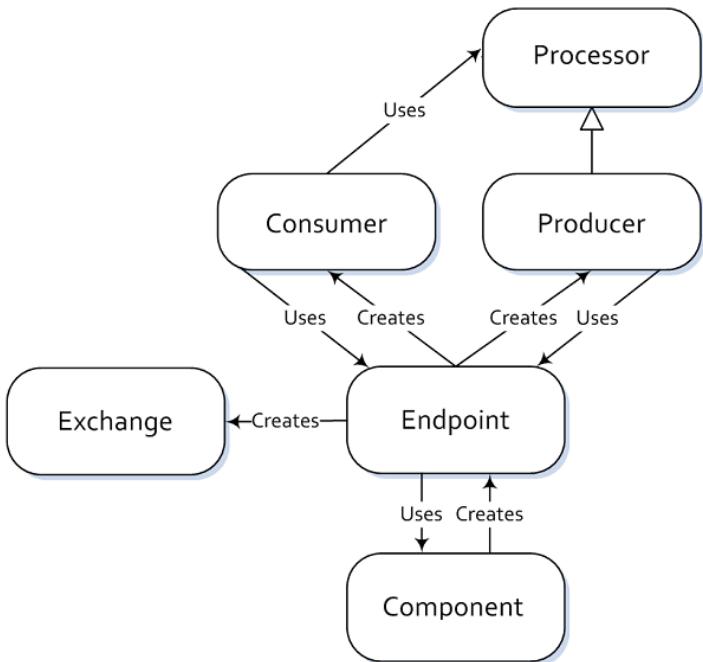


Figure 1.13 How endpoints work with producers, consumers, and an exchange

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, a `FileProducer` will write the message body to a file. A `JmsProducer`, on the other hand, will map the Camel message to a `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.

CONSUMER

A consumer is the service that receives messages produced by a producer, 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, we 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 using the routing engine.

In Camel there are 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.

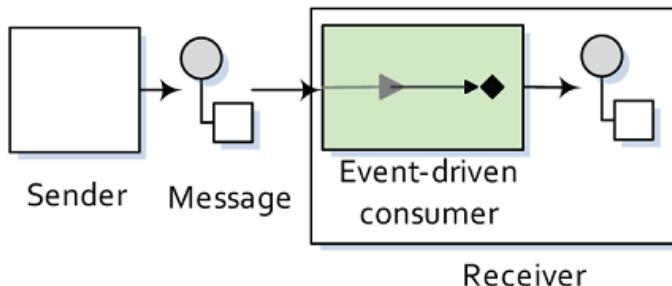


Figure 1.14 An event-driven consumer waits idle until a message arrives, at which point it wakes up and consumes the message.

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, like 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.

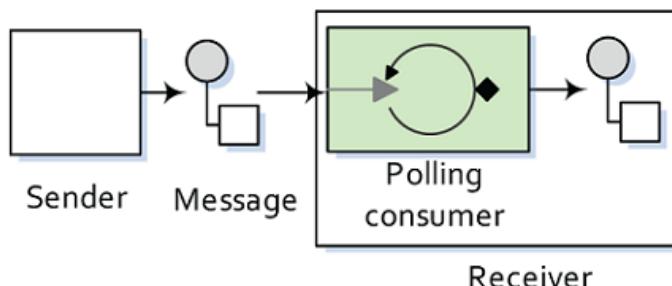


Figure 1.15 A polling consumer actively checks for new messages.

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 has

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 transports 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 really 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.

Listing 1.4 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")
                    .to("file:data/outbox");
            }
        });
        context.start();
        Thread.sleep(10000);
        context.stop();
    }
}
```

① Java DSL route

In this example, you first create a `CamelContext`, which is the Camel runtime. You then add the routing logic using a `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.

1.6 Summary

In this chapter you met Camel. You saw how Camel simplifies integration by relying on known 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 looked at 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 what the concepts behind Camel are. 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, we'll explore Camel's features and give you 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 that you get the main concepts from each chapter, from now on we'll present you with a number of best practices and key points in the summary.
- In the next chapter, we'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 enterprise integration patterns (EIPs)

One of the most important features of Camel is routing; without it, Camel would essentially be a library of transport connectors. In this chapter, we'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 different 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. This effectively means that the input and output queues are unaware of the conditions in between them. The conditional logic is decoupled from the message consumer and producer.

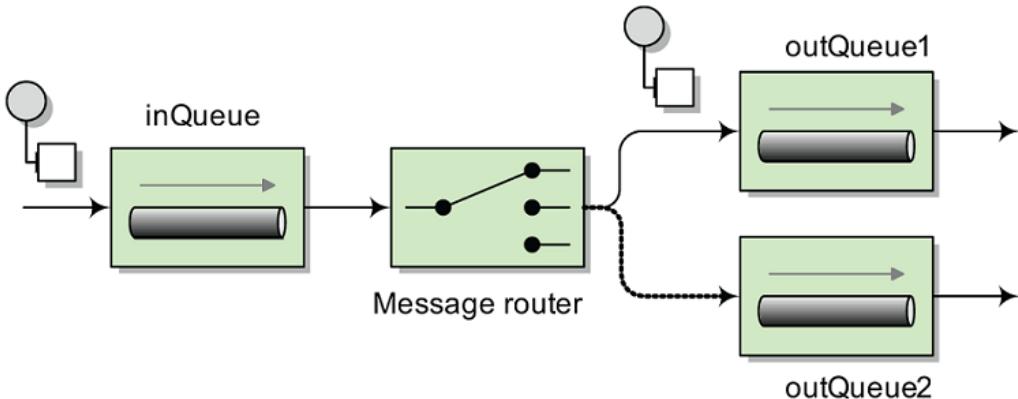


Figure 2.1 A message router consumes messages from an input channel and, depending on a set of conditions, sends the message to one of a set of output channels.

In Apache 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 some system, or even creating the message itself. This message then flows through a processing component, which could be an enterprise integration pattern (EIP), a processor, an interceptor, or some other 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.

In this chapter, we'll first introduce the fictional company that we'll 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) using Camel's endpoints. Following this, we'll look in depth at the Java-based domain-specific language (DSL) and the XML-based configuration format for creating routes. We'll also give you a glimpse of how to design and implement solutions to enterprise integration problems 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 that we'll use 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, they've changed the way they receive orders several times. Initially, orders were placed by uploading comma-separated value (CSV) files to an FTP

server. The message format was later changed to XML. Currently they provide 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, they 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.

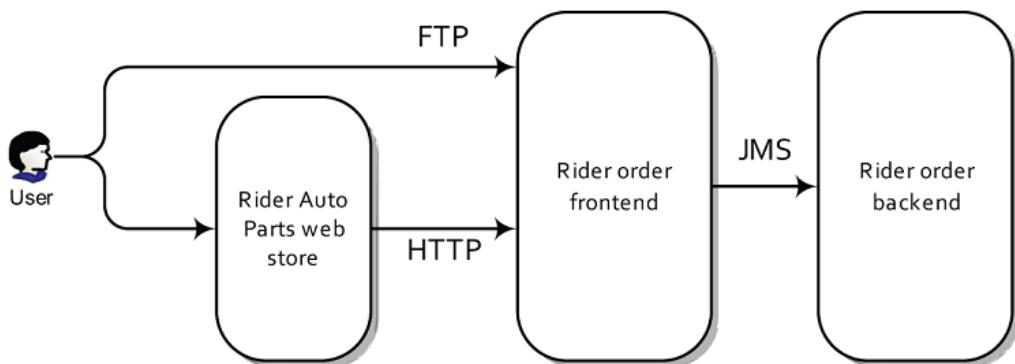


Figure 2.2 A customer has two ways of submitting orders to the Rider Auto Parts order-handling system: either by uploading the raw order file to an FTP server or by submitting an order through the Rider Auto Parts web store. All orders are eventually sent via JMS for processing at Rider Auto Parts.

Rider Auto Parts faces a pretty common problem: over years of operation, they have 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 their current requirements and new functionality using Camel.

As a first assignment, you'll need to implement the FTP module in the Rider order frontend system. Later in the chapter, you'll see how backend services are implemented too. Implementing the FTP module will involve 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'll need to understand how to communicate over FTP and JMS using Camel's endpoints. To complete the entire assignment, you'll need to understand routing with the Java DSL. Let's first take a look at how you can 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. In this section, we're going to explain how you can 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 make Camel easy to use is the endpoint URI. With an endpoint URI you can identify the component you want to use and how 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.

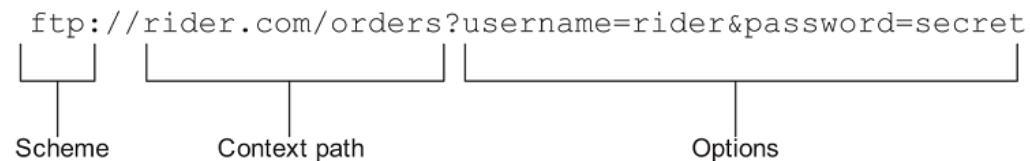


Figure 2.3 A Camel endpoint URI consists of three parts: a scheme, a context path, and a list of options.

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

The context path of `rider.com/orders` tells the `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 plaintext is not usually a good idea! You can find out how to use encrypted passwords in chapter 14.

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

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>2.17.1</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'll cover all the details in chapter 6. For now, though, we're just going to cover 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?

JMS (Java Message Service) 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, like 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, where each message has only one consumer. *Topics* operate on a publish/subscribe scheme; a single message may be delivered to many consumers if they have subscribed to the topic.

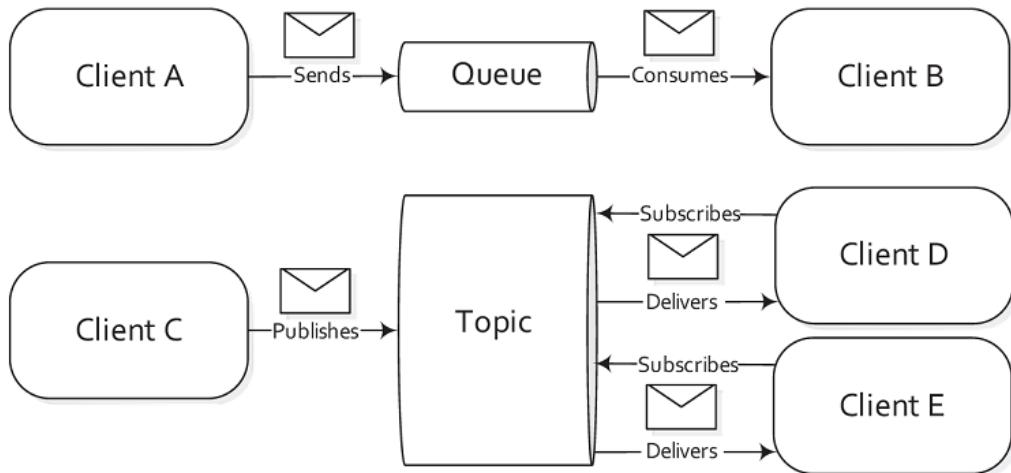


Figure 2.4 There are two types of JMS destinations: queues and topics. The queue is a point-to-point channel, where each message has only one recipient. A topic delivers a copy of the message to all clients who have subscribed to receive it.

JMS also provides a `ConnectionFactory` that clients (like 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, Dejan Bosanac, and Rob Davies, available from Manning Publications.

So 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 very 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. In order to use these, you'll need to add some 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>2.17.1</version>
</dependency>
```

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

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-all</artifactId>
  <version>5.13.1</version>
</dependency>
```

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

USING URIS TO SPECIFY THE DESTINATION

Once 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

```
jms:queue:incomingOrders
```

This is pretty 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 have omitted 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 over 70 options, many of which are only used in specific JMS scenarios. Camel always tries to provide built-in defaults that fit most cases, and you can always find out what the default values are 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 some messages!

2.3 Creating routes in Java

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

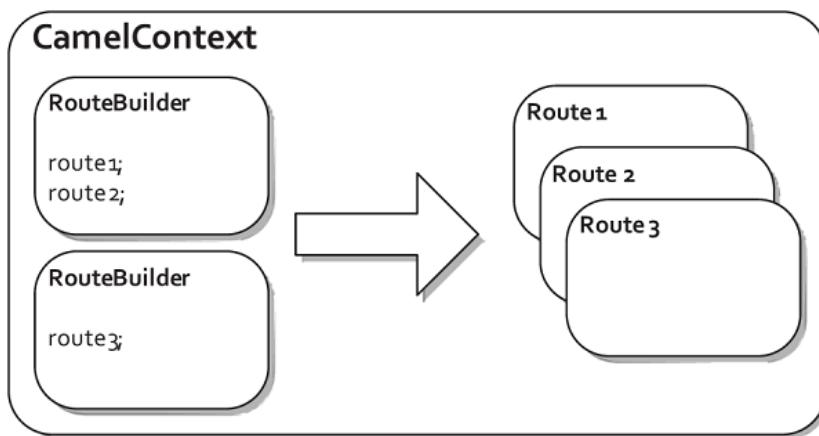


Figure 2.5 `RouteBuilders` are used to create routes in Camel. Each `RouteBuilder` can create multiple routes.

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

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

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

This means that 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 a `RouteBuilder` and the Java DSL to create simple routes. Once you know that, 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 the `RouteBuilder`

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

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

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

You then need to add the class to the `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 the `CamelContext`, like this:

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

Within the `configure` method, you define your routes using the Java DSL. We'll discuss 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 have downloaded the source code from the book's website 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. You can find setup instructions for Eclipse in Appendix A.

NOTE Eclipse is a popular open source IDE that you can find at <http://eclipse.org> . During the book's development, we used Eclipse 4.5.0. You can certainly use other Java IDEs as well or even no IDE, but it does make it a lot easier! Feel free to skip to the next section if you don't want to see the IDE-related setup.

Once 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-space in Eclipse) in the `configure` method, you'll be presented with a number of methods. To start a route, you should use the `from` method.

```

import org.apache.camel.CamelContext;
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
            }
        });
        context.start();
    }
}

```

The screenshot shows the Eclipse IDE interface with the Java code above. A tooltip is displayed over the `from` method call in the `configure` block. The tooltip contains the following information:

- Creates a new route from the given URI input**
- Parameters:** uri the from uri
- Returns:** the builder

Below the tooltip, a list of other `from` method variations is shown, including:

- `fromBodyAs(Class<?> type) : ValueBuilder - Build`
- `finalize() : void - Object`
- `from(Endpoint endpoint) : RouteDefinition - Route`
- `from(Endpoint... endpoints) : RouteDefinition - Ro`
- `from(String uri) : RouteDefinition - RouteBuilder`
- `from(String... uris) : RouteDefinition - RouteBuild`
- `fromF(String uri, Object... args) : RouteDefinition`
- `getClass() : Class<?> - Object`
- `getContext() : ModelCamelContext - RouteBuild`
- `getErrorHandlerBuilder() : ErrorHandlerBuilder - B`
- `getRestCollection() : RestsDefinition - RouteBuild`

At the bottom of the tooltip, it says "Press 'Tab' from proposal table or click for focus".

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 a FTP endpoint URI to connect to 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 a number of different 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 The Java DSL

Domain-specific languages (DSLs) are computer languages that target a specific problem domain, rather than a general purpose domain like most programming languages.

For example, you have probably used the regular expression DSL to match strings of text and found it to be a very 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* (*aka fluent builders*). When using a fluent interface, you build up objects with methods that perform an operation and then return the current object instance; another method is then invoked on this object instance, and so on.

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

Camel's domain is enterprise integration, so the Java DSL is essentially a set of fluent interfaces that contain methods named after terms from the EIP book. In the Eclipse editor, take a look at what is 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 EIPs—the Enricher and Recipient List—and there are many others that we'll discuss later.

```
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("ftp://rider.com/orders?username=rider&password=secret");
            }
        });
        context.start();
        Thread.sleep(10000);
        context.stop();
    }
}
```

The `ContentEnricher EIP` enriches an exchange with additional data obtained from a `resourceUri` using a `org.apache.camel.PollingConsumer` to poll the endpoint.

The difference between this and `enrich(String)` is that this uses a consumer to obtain the additional data, whereas `enrich` uses a producer.

This method will **block** until data is available, use the method with `timeout` if you do not want to risk waiting a long time before data is available from the `resourceUri`.

Parameters:

- `resourceUri` URI of resource endpoint for obtaining additional data.
- `aggregationStrategy` aggregation strategy to aggregate input data and additional data.

Returns:

- the builder

See Also: Press 'Tab' from proposal table or click for focus

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 listing 2.1.

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"
                    + "?username=rider&password=secret")
                    .to("jms:incomingOrders");
            }
        });
        context.start();
        Thread.sleep(10000);
        context.stop();
    }
}
```

① Java statement that forms a route

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

As you can see, this listing includes a bit of boilerplate setup and configuration, but the actual 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, where the output of the consumer is fed into the producer as input. This is depicted in figure 2.8.

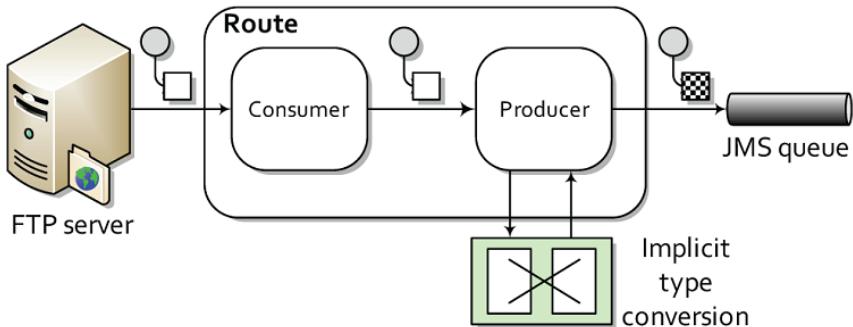


Figure 2.8 The route shown in listing 2.1 forms a simple pipeline. The output of the FTP consumer is fed into the input of the JMS producer. The payload conversion from file to JMS message is done automatically.

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 `TypeConverter` 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 would be really 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 is a pretty 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 out 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 effectively 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, like the `FileComponent` and the `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>.

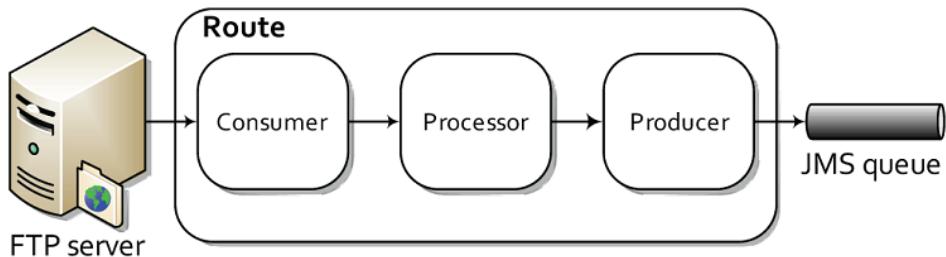


Figure 2.9 With a processor in the mix, the output of the FTP consumer is now fed into the processor, and then the output of the processor is fed into the JMS producer.

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 Groovy, Scala, and, as we'll discuss next, XML.

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. However, 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, I know many system administrators who handily write up Camel routes to solve integration problems but have never used Java in their life. The XML configuration also makes nice graphical tooling¹ possible that has round trip capabilities – meaning that you can edit both the XML and graphical representation of a route and both are

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

kept in sync. Round trip tooling with Java is not impossible but its a seriously hard thing to do so none is yet available.

At the time of writing, you can write XML routes in two Inversion of Control (IoC) Java containers: Spring and OSGi Blueprint. An IoC framework essentially allows to you “wire” beans together to form applications. This wiring is typically done through an XML configuration file. In this section, we’ll give 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. Also OSGi Blueprint is covered nicely in *OSGi in Action*, by Richard S. Hall, Karl Pauls, Stuart McCulloch, and David Savage.

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

2.4.1 Bean injection and Spring

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

Consider an application that prints out 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, 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 names the bean myGreeter
2. Create an instance of GreetMeBean and names the bean greetMeBean
3. Set the reference of the greeter property of the GreetMeBean to the bean named myGreeter

This configuring of beans is called *wiring*.

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

Here is 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 the `DanishGreeter` in instead, you'd have seen something like this on the console:

```
Davs janstey
```

This example may seem pretty simple, but it should give you an understanding of what Spring and, more generally, an IoC container, really is.

So how does Camel fit into this? Essentially, Camel can be configured as if it were another bean. For instance, you configured the JMS component to connect to an ActiveMQ broker in section 2.2.2, but 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 like `"jms:incomingOrders"`, Camel will look up the `"jms"` bean and if it is of type `org.apache.camel.Component`, it will use that. So you don't have to manually add components to the `CamelContext` — a task which you did manually in section 2.2.2 for the Java DSL.

But where is the `CamelContext` defined in Spring? Well, to make things easier on the eyes, Camel utilizes Spring extension mechanisms to provide custom XML syntax for Camel concepts within the Spring XML file. To load up a `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 will automatically start a `SpringCamelContext`, which is a subclass of the `DefaultCamelContext` you used for the Java DSL. Also notice that you had 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 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="camelinaction.FtpToJMSRoute"/>
    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <routeBuilder ref="ftpToJmsRoute"/>
    </camelContext>
</beans>
```

You may have noticed that we're referring to the `camelinaction.FtpToJMSRoute` class as a `RouteBuilder`. In order 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 the `RouteBuilder` purely in XML. The following Spring XML does this.

Listing 2.3 A 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 using elements with names similar to ones used inside the Java DSL `RouteBuilder`. Notice that we actually had to modify the FTP endpoint URI to ensure it is valid XML. The ampersand character '`&`' used to define extra URI options is actually 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 very useful for testing. In chapter 6, you'll also 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 an additional processing step for testing.

ADDING A PROCESSOR

Adding additional processing steps is simple, as in the Java DSL. Here you'll add a custom processor like 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 it in your XML DSL route as follows:

```
<bean id="downloadLogger" class="camelaction.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 is only one message file named message1.xml in the src/data directory, this will output 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'll 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, simply add an additional `route` element within the `camelContext` element.

For example, move the `DownloadLogger` bean 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 are consuming the message from the `incomingOrders` queue in the second route. So, the downloaded message will be printed after the order is sent to 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, like value builders (for building expressions and predicates ²), aren't available in the XML DSL. On the other hand, using Spring 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 – meaning that you can edit both the XML and graphical representation of a route and both are kept in sync.

A common compromise is to use both Spring and the Java DSL, which is one of the topics we'll cover next.

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 of using Camel. In fact, it's the most frequent usage of Camel.

You saw before that you can explicitly tell the Spring `CamelContext` what route builders to load. You can do this by using the `routerBuilder` 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.

² See appendix B for more information on expressions and predicates.

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>camelaction.routes</package>
  </packageScan>
</camelContext>
```

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

You can even be a bit more picky 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 is 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 Spring route builders within the `camelaction.routes` package that have the `@Component` annotation.

Under the hood, some of Camel's components, like the JMS component, are built on top of abstraction libraries from Spring. It makes sense that 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 like `activemq:myActiveMQQueue` or `wmq:myWebSphereQueue`.

Endpoints can also be defined using Camel's Spring XML extensions. For example, you can break out the FTP endpoint for connecting to Rider Auto Parts' legacy order server into an `<endpoint>` element which is highlighted in bold below:

```
<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 would be 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 its often easier to read them if you break them up over several lines. See the previous route with the endpoint URI options broken out 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 out 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, some other 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 the `CamelContext` by using the following line:

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

Now the `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 a `camelContext` as follows:

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

```
<routeContextRef ref="ftpToJms"/>
</camelContext>
```

If you import the `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`.

ADVANCED SPRING CONFIGURATION OPTIONS

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

- Pluggable bean registries are discussed in chapter 4.
- The Tracer and Delay mechanisms are covered in chapter 16.
- Custom class resolvers, tracing, fault handling and startup are mentioned in chapter 15.
- The configuration of interceptors is covered in chapter 9.

2.5 Endpoints revisited

Back 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 some more advanced endpoint configurations.

2.5.1 Sending to dynamic endpoints

Endpoint URIs like this JMS one shown back in section 2.2.2 are evaluated just once when Camel starts up. So they are essentially 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 needed to determine these names at runtime? Static endpoint URIs provided to the `to` method will be of no use in this case as they are only evaluated once at startup. Camel provides an additional DSL method for this: `toD`.

For example, say you wanted to make the endpoint URI point to a destination name stored as a message header? The following would do this:

```
.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. So if "myDest" is "incomingOrders" then the resultant endpoint URI will be "jms:queue:incomingOrders", like we had before in the static case. If you were wondering about the "simple" language, it is 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 also there is a complete reference in appendix B.

While the simple language is the default expression language used with the `toD` method, you can also use any of the other expression languages in Camel. Let's say we wanted to append the customer name to the queue we are sending to. We could use Xpath to grab an attribute from the XML message that has this customer name:

```
.toD("jms:${header.myDest}+language:xpath:/order/@customer");
```

Likewise in the XML DSL we would have:

```
<toD uri="jms:${header.myDest}+language:xpath:/order/@customer"/>
```

Here we use a `+` to append an additional expression to the result, where "language" specifies we want something other than a simple expression. We then add "`xpath:/order/@customer`" to actually specify the Xpath expression that points to the customer name attribute we are looking for. Now, for an incoming message with body something like:

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

Our `toD` method will evaluate to a endpoint URI of "`jms:queue:incomingOrders.honda`".

To run these examples yourself, go to the `chapter2/ftp-jms` directory in the book's source code and run these Maven commands:

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

2.5.2 Using property placeholders in endpoint URIs

Rather than having hard coded endpoint URIs, Camel allows you to use property placeholders in the URIs to replace the dynamic parts. By dynamic here we mean that values will be replaced when Camel starts up, not on every new message like 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, 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 hard coding them.

USING THE PROPERTIES COMPONENT

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

- It is built in the camel-core JAR, which means it can be leveraged without the need for Spring or any third-party framework.

- It can be used in all the DSLs, such as the Java DSL, and is not limited to Spring XML files.
- It supports using placeholders in property files.

NOTE 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 section 14.1.

To ensure that the property placeholder is loaded and in use as early as possible, you have to configure the PropertiesComponent when the 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 run try this example 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:

```
<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 trace="true" id="camel" 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 the `camelContext`, as follows:

```
<camelContext trace="true" id="camel" 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 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 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 trace="true" id="camel" 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" ①
                 location="classpath:rider-test.properties"/>

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

    <endpoint id="myDest" uri="jms:${myDest}"/> ②

    <route>
        <from uri="file:src/data?noop=true" />
        <to ref="myDest" /> ③
    </route>

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

- ① Loads properties from external file
- ② Defines endpoint using property placeholders
- ③ Refers to endpoint in route

To use Spring property placeholders, you must declare the `<context:property-placeholder>` tag where you refer to a `properties` bean ① that will load the properties file from the classpath.

In the `camelContext` element, you define an endpoint ② 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 ③ 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 using the following Maven goal:

```
mvn test -Dtest=SpringFtpToJMSWithSpringPropertyPlaceholderTest
```

The Camel Properties component versus Spring property placeholders

The Camel Properties component is more powerful than the Spring property placeholder mechanism. The latter only works 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, Blueprint OSGi XML, Groovy, and Scala. On top of that, you can declare the placeholders anywhere in the route definitions.

2.5.3 Using raw values in endpoint URLs

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 is using reserved characters. You could encode these reserved characters but this would make things less readable (not that you'd want your password more readable – it is just an example!). Camel's solution is to allow “raw” values in endpoint URLs which don't count toward URI validation. So, 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 URLs

You've heard the Camel registry mention a few times now but have not seen it actually in use. Well, we don't dive into detail about the registry here – that is covered in Chapter 4. We will however, show a very common syntax in endpoint URLs related to the registry. Basically any

time a Camel endpoint requires an object instance as an option value, you can refer to one in the registry but using the “#” syntax. For example, say we only wanted to fetch CSV order files from the FTP site. We 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 we 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 us, you’re ready to tackle more advanced routing topics 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. This was intentional. We wanted to make sure you had 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 will be introduced throughout the book, and in chapter 5 we’ll be covering the most complex EIPs. The complete list of EIPs supported by Camel is available from the Camel website (<http://camel.apache.org/enterprise-integration-patterns.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 implies, 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, 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. That means 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 obviously 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.

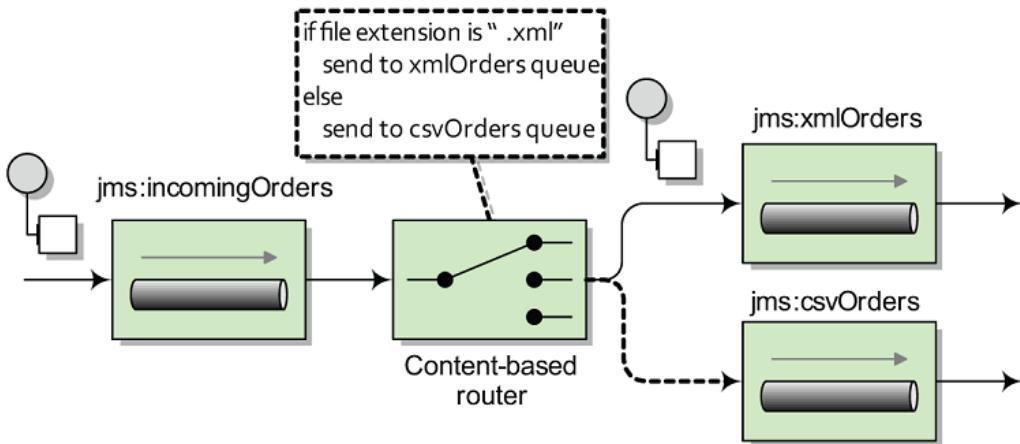


Figure 2.10 The CBR routes messages based on their content. In this case, the filename extension (as a message header) is used to determine which queue to route to.

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 only has 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. There are many different expression languages to choose from in Camel, some of which include Simple, EL, JXPath, Mvel, OGNL, PHP, BeanShell, JavaScript, Groovy, Python, Ruby, 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 the `RouteBuilder`, you can start by using the `header` method, which returns an expression that will evaluate to the header value. For example, `header("CamelFileName")` will create an expression that will resolve to the value of the `CamelFileName` header on the incoming exchange. On this expression you can invoke a number of methods to create a predicate. So, 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 here.

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")
            .choice()
                .when(header("CamelFileName").endsWith(".xml"))
                    .to("jms:xmlOrders")
                .when(header("CamelFileName").endsWith(".csv"))
                    .to("jms:csvOrders");
    }

    from("jms:xmlOrders")
        .log("Received XML order: ${header.CamelFileName}")
        .to("mock:xml");

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

① Content-based router

② Test routes that print message content

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 will consume two order files in the `chapter2/cbr/src/data` directory and output 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 out messages 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 like we 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. We could also replace our 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 we 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 B to cover it fully.

You use these routes to test that the router ① is working as expected. Route-testing techniques, like use of the Mock component, will be discussed in chapter 9.

Of course we can also form an equivalent CBR using the XML DSL as shown in Listing 2.5. Other than being in XML rather than Java, the main difference to note here is that you use a Simple expression instead of the Java-based predicate ③. 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" />
```

①

```

<choice>
  <when>
    <simple>${header.CamelFileName} ends with 'xml'</simple> ③
    <to uri="jms:xmlOrders" />
  </when>
  <when>
    <simple>${header.CamelFileName} ends with 'csv'</simple>
    <to uri="jms:csvOrders" />
  </when>
</choice>
</route>

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

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

```

- ① Content-based router
- ② Test routes that print message content
- ③ Simple expression used instead of Java-based predicate

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 will see output similar to that of the Java DSL example.

USING THE OTHERWISE CLAUSE

One of Rider Auto Parts' customers sends CSV orders with the .csl extension. Your current route only handles .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 simple `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. Really, 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>${header.CamelFileName} regex '^.*xml$'</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 will consume four order files in the `chapter2/cbr/src/data_full` directory and output the following:

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

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

ROUTING AFTER A CBR

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, like 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. Now, after each destination with the `choice`, the message will be routed to the `continuedProcessing` queue as well. This is illustrated in figure 2.11.

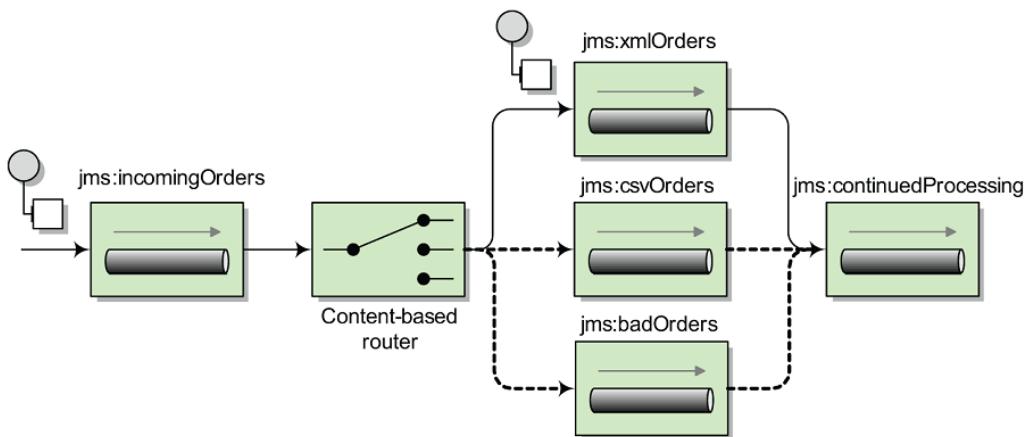


Figure 2.11 By using the `end` method, you can route messages to a destination after the CBR.

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 this 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 only be sent 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>${header.CamelFileName} regex '^.*xml$'</simple>
            <to uri="jms:xmlOrders"/>
        </when>
        <when>
            <simple>${header.CamelFileName} regex '^.*(csv|cs1)$'</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—their QA department has expressed the need to be able to send test orders into the live web frontend 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 only pass through the filter if a certain condition is met. Messages failing the condition will be dropped.

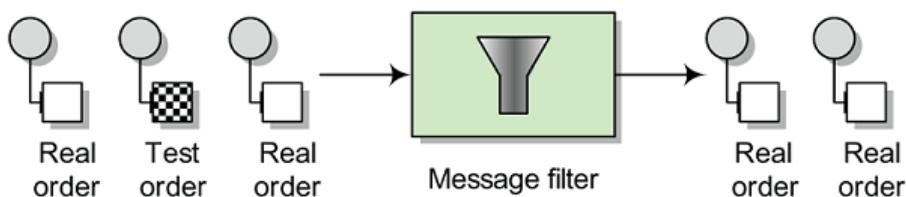


Figure 2.12 A Message Filter allows you to filter out uninteresting messages based on some condition. In this case, test messages are filtered out.

Let's see how you can implement this using Camel. Recall that the web frontend that Rider Auto Parts uses only sends orders 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 will output the following on the command line:

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

You'll only receive 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 `true` for orders that don't have the `test` attribute.

As you saw back in section 2.4.2, when the XML DSL is used, you cannot use an anonymous inner class for a processor. You must name the `Processor` class and add a bean entry in the Spring XML file. So 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>
```

The flow remains the same as in the Java DSL version of this route, but here you reference the processor as `orderLogger`, which is defined as a bean entry in the XML file.

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 we've looked at only sent messages to a single destination. Next we'll take a look at how you can 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 different 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 that they could improve 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 OK 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.

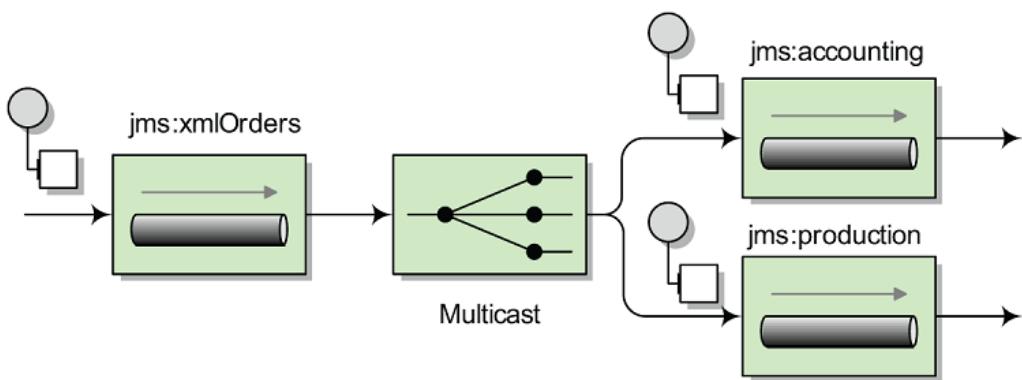


Figure 2.13 A multicast sends a message to a number of specified recipients.

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=OrderRouterWithMulticast
mvn clean test -Dtest=SpringOrderRouterWithMulticast
```

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 wanted to send them in parallel?

PARALLEL MULTICASTING

Sending messages in parallel 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 will set 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, please see chapter 13.

The equivalent route in XML DSL is as follows:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  ...
<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 this 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, essentially 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 bit 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 pretty static solution, because changing the destinations means changing the route. Let's see how you can 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 are not 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.

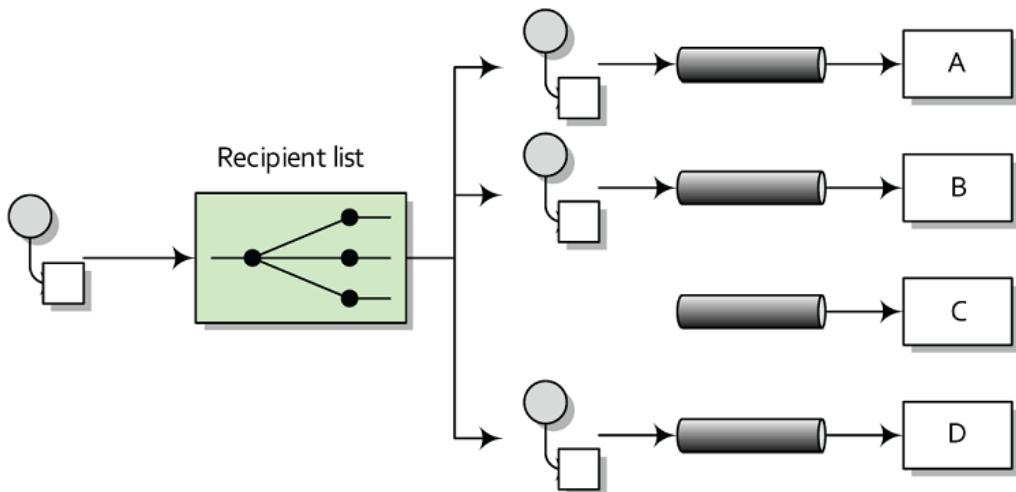


Figure 2.14 A recipient list inspects the incoming message and determines a list of recipients based on the content of the message. In this case, the message is only sent to the A, B, and D destinations.

Camel provides a `recipientList` method for implementing the Recipient List EIP. For example, the following route will take 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 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 or not. A simple solution could be to call out to a custom bean to do this:

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

Where `RecipientsBean` is a simple utility class like:

```
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 very similar layout:

```

<bean id="recipientsBean" class="camelinaction.RecipientsBean"/>

<camelContext id="camel" 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>
    ...

```

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.

It's common for recipients to not be embedded in the message as headers or parts of the body, and using a custom processor for this case is perfectly functional, but not very nice. In using a custom processor, you have to manipulate the exchange and message APIs directly. Fortunately, Camel supports a better way of implementing a recipient list.

RECIPIENT LIST ANNOTATION

Rather than using the `recipientList` method in the DSL, you can add a `@Recipient-List` 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 only gets invoked, however, 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 very similar layout:

```
<bean id="annotatedRecipientList" class="camelaction.AnnotatedRecipientList"/>

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

```

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 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'll 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 will output 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 it's useful and necessary to inspect messages as they flow through a system. 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 a incoming message to the console or append it to a file. Here is 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 pretty 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.

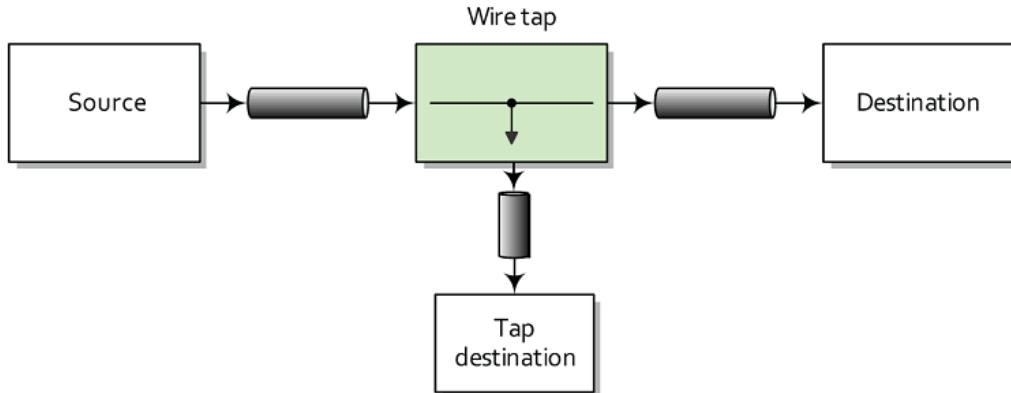


Figure 2.15 A wire tap is a fixed recipient list that sends a copy of a message traveling from a source to a destination to a secondary destination.

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`. This means that 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 this command:

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

So what can you do with a tapped message? A number of things could be done at this point:

- You could print the information to the console like 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 pretty 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 12.

2.7 Summary and best practices

In this chapter, we've covered one of the core abilities 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, booking a flight or hotel room, messages are being routed behind the scenes using some sort of router.
- *Use Apache Camel for routing messages.* Camel is primarily a message router that allows to you 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.* In section 2.4.3 we described our favorite way to write Camel applications, which is to boot up the 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 actual business problem rather than the integration architecture.

In the coming chapters we'll build on this foundation to show you things like data transformation, error handling, testing, sending data over other transports, and more. In the next chapter, we'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. They were topics you absolutely needed to know to use Camel. In this next part, we'll cover in depth the core features of Camel. 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, we'll look at how you can transform this data to other formats using Camel.
- Camel has great support for integrating beans into your routing applications. In chapter 4 we'll look at the many ways beans can be used in Camel applications.
- In complex enterprise systems, lots of things can go wrong. This is why Camel features an extensive set of error-handling abilities. In chapter 5 we'll discuss these in detail.
- In chapter 6 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.
- 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 7 covers the most heavily used components that ship with Camel.
- The last chapter 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 8, we'll look at several of the more complex EIPs.

3

Transforming data with Camel

This chapter covers

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

In the previous chapter, we covered routing, which is the single most important feature any integration kit must provide. In this chapter, we'll take a look at the second most important feature: data or message transformation.

Just as in 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 is a key feature 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 how you can transform data into any custom format you may have. Then we'll look at some Camel components that are specialized for transforming XML data and other well-known data formats. We'll end the chapter by looking into Camel's type-converter mechanism, which supports implicitly and explicitly type coercing.

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

3.1 Data transformation overview

Camel provides many techniques for data transformation, and we'll cover them shortly. But first we'll 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 a `java.lang.String` is transformed into a `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, where 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.

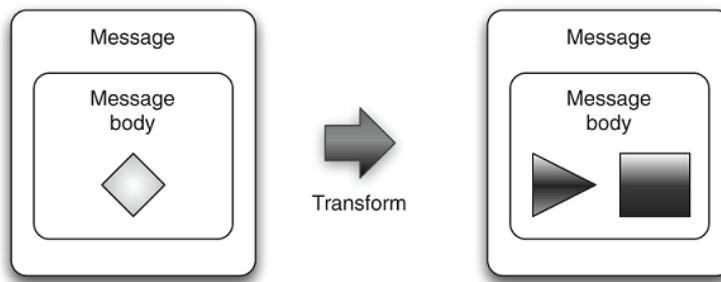


Figure 3.1 Camel offers many features for transforming data from one form to another.

Camel has many data-transformation features. We'll introduce them in the following section, and then look at them one by one. After reading this chapter, you'll have a solid understanding of how to use Camel to transform your data.

3.1.1 Data transformation with Camel

In Camel, data transformation typically takes places 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 using the Message Translator or the Content Enricher EIPs. This gives you the power to do data mapping using regular Java code. We'll 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'll dive into this in section 3.3. |
| Data transformation using data formats | Data formats are Camel transformers that come in pairs to transform data back and forth between well-known formats. |
| Data transformation using templates | Camel provides a range of components for transforming 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> . Type converters are covered in section 3.6. |
| 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. We'll cover writing custom components in chapter 8. |

In this chapter, we'll cover the first five of the data transformation methods listed in table 3.1. We'll leave the last one for chapter 8 because it only applies to writing custom components.

3.2 Transforming data using EIPs and Java

Data mapping is the process of mapping between two distinct data models, and it's 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 at first might seem elegant but that turns out to make things impossible.

In this section, we'll look at how you can map data using a Processor, which is a Camel API. Camel can also use 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.

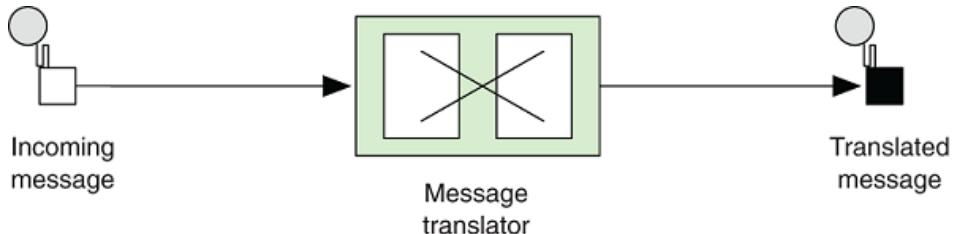


Figure 3.2 In the Message Translator EIP, an incoming message goes through a translator and comes out as a translated message.

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. 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 a Processor
- Using beans
- Using <transform>

We'll look at them each in turn.

TRANSFORMING USING A PROCESSOR

The Camel `Processor` is an interface defined in `org.apache.camel.Processor` with a single method:

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

The `Processor` is a low-level API where you work directly on the Camel `Exchange` instance. It gives you full access to all Camel's moving parts from the `CamelContext`, which you can obtain from the `Exchange` 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.

Listing 3.1 Using a Processor to translate from a custom format to 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
        String id = custom.substring(0, 9); ②
        String customerId = custom.substring(10, 19); ②
        String date = custom.substring(20, 29); ②
        String items = custom.substring(30); ②
        String[] itemIds = items.split("@"); ②
        StringBuilder csv = new StringBuilder(); ③
        csv.append(id.trim());
        csv.append(",").append(date.trim());
        csv.append(",").append(customerId.trim());
        for (String item : itemIds) {
            csv.append(",").append(item.trim());
        }
        exchange.getIn().setBody(csv.toString()); ④
    }
}
```

- ① Gets custom payload
- ② Extracts data to local variables
- ③ Maps to CSV format
- 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 the `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 Spring 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=yesterday"/>
        <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.

There are two scenarios where the Camel end user will have to decide among using these methods:

- 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 using `getIn`, and not to use `getOut` at all. This is the practice we use in this book.

Using a processor has one disadvantage: you're required to use the Camel API. In the next section, we'll look at 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.

Let's try using a bean instead of a 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, 9);
        String customerId = custom.substring(10, 19);
```

①

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

```

String date = custom.substring(20, 29);          ①
String items = custom.substring(30);
String[] itemIds = items.split("@");
StringBuilder csv = new StringBuilder();
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
}
}

```

- ① Extracts data to local variables
- 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. This means 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; we'll cover much more about using beans in chapter 4.

The actual 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 Spring 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=yesterday" />
        <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

```

It will generate 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 very 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 some text for HTML formatting by replacing all line breaks with a `
` tag. This can be done 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 what is known as 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 only allows connectivity 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 section 6 in 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 have 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 you can transform data using Spring XML.

TRANSFORMING USING `<transform>` FROM SPRING XML

Using `<transform>` from Spring XML is a bit different than from Java DSL because the XML DSL isn't as powerful. In Spring XML, 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="camelinaaction.HtmlBean"/> ①
<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>
```

- ① Does the transformation
- ② Invokes `toHtml` method on bean

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 very 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, MVEL, JavaScript, or Camel's own scripting language, called Simple (explained in some detail in appendix B). We won't go in detail on how to use the other scripting languages at this point, but the Simple language can be used to build strings using placeholders. It pretty much speaks for itself—I'm sure you'll understand what the following transformation does:

```
<transform>
  <simple>Hello ${body} how are you?</simple>
</transform>
```

You can try the Spring 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
```

They're located in the chapter3/transform directory and are named SpringTransformMethodTest and 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 where a message is enriched with data obtained from another resource.

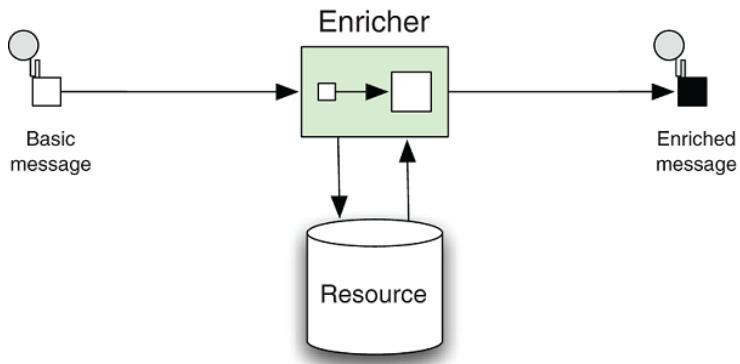


Figure 3.3 In the Content Enricher EIP, an existing message has data added to it from another source.

To help understand this pattern, we'll 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.

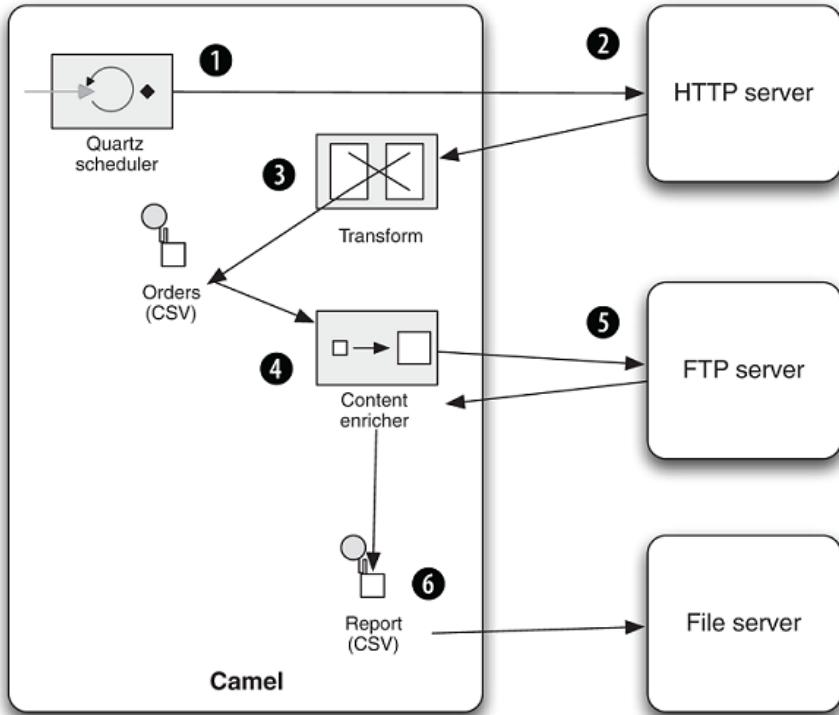


Figure 3.4 An overview of the route that generates the orders report, now with the content enricher pulling in data from an FTP server

In figure 3.4, 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 we dig into the code and see how to implement this, we 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 using a consumer.
- `enrich`—This method merges data retrieved from another source 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 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 only works 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.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 using Java code and return the merged result. This may sound a bit confusing, so let's see it 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

Listing 3.3 shows how you can use `pollEnrich` to retrieve the additional orders from the remote FTP server and aggregate this data with the existing message 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")
    .process(new OrderToCSVProcessor())
    .pollEnrich("ftp://riders.com/orders/?username=rider&password=secret",
        ①
        new AggregationStrategy() {
            public Exchange aggregate(Exchange oldExchange,
                Exchange newExchange) {
                if (newExchange == null) {
                    return oldExchange;
                }
                String http = oldExchange.getIn()
                    .getBody(String.class);
                ②
                String ftp = newExchange.getIn()
                    .getBody(String.class);
                ②
                String body = http + "\n" + ftp;
                oldExchange.getIn().setBody(body);
                ②
                return oldExchange;
            }
        })
    .to("file://riders/orders");
    ③
```

① Uses `pollEnrich` to read FTP file

- ② Merges data using AggregationStrategy
- 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 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 or not. If `newExchange` is `null`, there is no remote file to consume, and you just return the existing data. If there is 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.

`PollEnrich` uses a polling consumer to retrieve messages, and it offers three timeout modes:

- `pollEnrich(timeout = -1)`—Polls the message and waits until a message arrives. This mode will block until a message exists.
- `pollEnrich(timeout = 0)`—Immediately polls the message if any exists; otherwise `null` is returned. It will never wait for messages to arrive, so this mode will never block. This is the default mode.
- `pollEnrich(timeout > 0)`—Polls the message, and if no message exists, it will wait for one, waiting at most until the timeout triggers. This mode will potentially block.

It's a best practice to either use `timeout = 0` or to assign a timeout 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 Spring XML; it's a bit different than when using the Java DSL.

ENRICHING USING ENRICH

`Enrich` is used when you need to enrich the current message with data from another source using request-reply messaging. A prime example would be to enrich the current message with the reply from a web service call. But we'll look at another example, using Spring XML to enrich the current message using the TCP transport:

```
<bean id="quoteStrategy"
    class="camelaction.QuoteStrategy"/>          ①
<route>
    <from uri="activemq:queue:quotes"/>
    <enrich url="netty4:tcp://riders.com:9876?textline=true&sync=true"
        strategyRef="quoteStrategy"/>
    <to uri="log:quotes"/>
</route>
```

- ① Bean implementing AggregationStrategy

Here you use the Camel `netty4` component for the TCP transport, configured to use request-reply messaging by using `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 actual 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 actual 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 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 using XSLT stylesheets
- *XML marshaling*—For marshaling and unmarshaling objects to and from XML

Both of these will be covered in following sections.

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 you could use it; this route is also illustrated in figure 3.5.

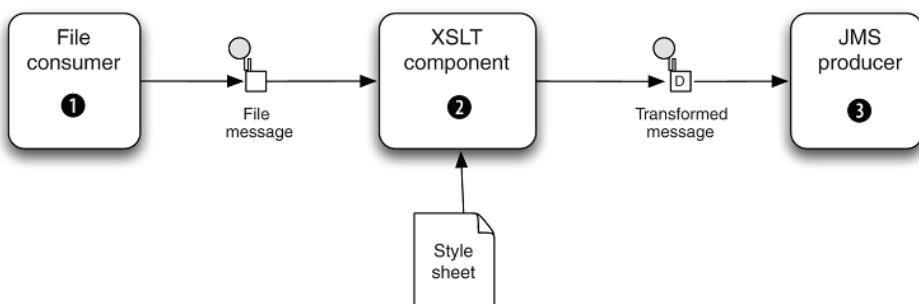


Figure 3.5 A Camel route using an XSLT component to transform an XML document before it's sent to a JMS queue

```
from("file:///rider/inbox")
    .to("xslt://camelaction/transform.xsl")
    .to("activemq:queue:transformed")
```

The file consumer picks up new files and routes them to the XSLT component, which transforms the payload 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 |
|--------------------|---|---|
| <none> | <code>xslt://camelaction/transform.xsl</code> | If no prefix is provided, Camel loads the resource from the classpath |
| <code>file:</code> | <code>xslt://file:///rider/config/transform.xml</code> | Loads the resource from the filesystem |
| <code>http:</code> | <code>xslt://http://rider.com/styles/transform.xsl</code> | Loads the resource from an URL |

Let's leave the XSLT world now and take a look at how you can 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*. We'll cover data formats in full detail in section 3.4, but we'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. Let's look at 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"/>
    </dataFormats>
```

1

```

<route>
    <from uri="direct:foo"/>
    <marshal ref="myXstream"/>
    <to uri="activemq:queue:foo"/>
</route>
</camelContext>

```

- ① Specifies XStream data format
- ② Transforms to XML

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="activemq:queue:foo"/>
</route>

```

The same route is a bit shorter to write in the Java DSL, because you can do it with one line per route:

```
from("direct:foo").marshal().xstream().to("uri:activemq: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="activemq: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

JAXB (Java Architecture for XML Binding) 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 often 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 com.acme.order;
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
@XmlAccessorType(XmlAccessType.FIELD)
public class PurchaseOrder {
    @XmlAttribute
    private String name;
    @XmlAttribute
    private double price;
    @XmlAttribute
    private double amount;
}
```

① PurchaseOrder class is JAXB annotated

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="4995" amount="1"/>
```

Listing 3.6 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"/>
        <marshal ref="jaxb"/> ②
        <to uri="activemq:queue:order"/>
    </route>
    <route>
        <from uri="activemq:queue:order"/>
        <unmarshal ref="jaxb"/> ③
        <to uri="direct:doSomething"/>
    </route>

```

```
</camelContext>
```

- ① Declares JAXB data format
- ② Transforms from model to XML
- ③ Transforms from XML to model

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 the context path. 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 of them 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, EDI, 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, meaning that one is capable of reversing what the other has done, as illustrated in figure 3.6.

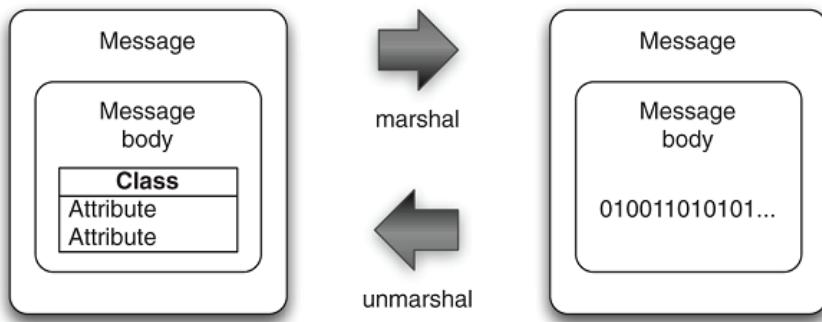


Figure 3.6 An object is marshaled to a binary representation; unmarshal can be used to get the object back.

We touched on data formats in section 3.3, where we covered XML transformations. This section will cover data formats in more depth and using other data types than XML, such as CSV and JSON. We'll even look at how you can 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, as listed in table 3.3.

Table 3.3 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 using Apache Avro. |
| Barcode | Barcode image | camel-barcode | Can convert a String into a barcode image and back again. |
| Base64 | Base64 string | camel-base64 | Can encode and decode into a base64 string. |
| BeanIO | XML, CSV, delimited, fixed length | camel-beanio | Uses BeanIO for binding various formats to and from Java objects |
| Bindy | CSV, FIX, fixed length | camel-bindy | Binds various data models to model objects using annotations |
| Boon | JSON | camel-boon | Transforms to and from JSON using the Boon library |
| Castor | XML | camel-castor | Uses Castor for XML binding to and from Java objects |

| | | | |
|---------------|------------------------|-------------------------|---|
| Crypto | Any | camel-crypto | Encrypts and decrypts data using the Java Cryptography Extension |
| CSV | CSV | camel-csv | Transforms to and from CSV using the Apache Commons CSV library |
| Flatpack | CSV | camel-flatpack | Transforms to and from CSV using the FlatPack library |
| Gson | JSON | camel-gson | Transforms to and from JSON 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 |
| iCal | iCalendar | camel-ical | Transforms between iCalendar data and ical4j model objects. |
| 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 using the ultra-fast Jackson library |
| JiBX | XML | camel-jibx | Uses the JiBX library for XML binding to and from Java objects |
| PGP | Any | camel-crypto | Encrypts and decrypts data using PGP. |
| Protobuf | XML | camel-protobuf | Transforms to and from XML 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. |
| TidyMarkup | HTML | camel-tagsoup | Tidies up HTML by parsing ugly HTML and returning it as pretty well-formed HTML |
| UniVocity | CSV, fixed-length, TSV | camel-univocity-parsers | Transforms to and from CSV, fixed-length, or TSV formats using the UniVocity Parsers project. |
| XmlBeans | XML | camel-xmlbeans | Uses XmlBeans for XML binding to and from Java objects |
| XmlJson | JSON | camel-xmljson | Transforms between XML and JSON. |

| | | | |
|-------------|----------|-------------------|---|
| XmlRpc | XML-RPC | camel-xmlrpc | Transforms between XML formatted to XML-RPC specification requirements and Apache XML-RPC 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 using the XStream library |
| Zip | Any | camel-core | Compresses and decompresses messages; it's most effective when dealing with large XML- or text-based payloads |
| Zip File | Zip File | camel-zipfile | Compresses and decompresses zip files. |

As you can see, Camel provides 31 data formats out of the box. We've picked 3 to cover in the following section. They're among the most commonly used, and what you learn about those will also apply for the remainder of the data formats. You can read more about all these data formats at the Camel website (<http://camel.apache.org/data-format.html>).

3.4.2 Using Camel's CSV data format

The camel-csv data format is capable of transforming to and from CSV format. It leverages Apache Commons CSV to do the actual 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("activemq: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 Spring 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="activemq:queue.csv.record"/>
        </split>
    </route>
</camelContext>
```

```
</split>
</route>
</camelContext>
```

The noticeable difference is how 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 B), which knows how to do that.

NOTE The Splitter EIP is fully covered in section 5.1 of this book.

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 a bit 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 | Map<String, Object> | OutputStream | Contains a single row in CSV format |
| marshal | List<Map<String, Object>> | OutputStream | Contains multiple rows in CSV format where each row is separated by \n(newline) |
| unmarshal | InputStream | List<List<String>> | Contains a List of rows where each row is another List of fields |

One problem with camel-csv is that it uses generic data types, such as `Maps` or `Lists`, to represent CSV records. Often you'll already have model objects to represent your data in memory. Let's look at how you can use 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 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.

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") ①
public class PurchaseOrder {
    @DataField(pos = 1) ②
    private String name;
    @DataField(pos = 2, precision = 2) ②
    private BigDecimal price;
    @DataField(pos = 3) ②
    private int amount;
}
```

① Maps to CSV record

② Maps to column in CSV record

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 outputted 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 we look at how to use Bindy in Camel routes, we need to take a step back and look at the data types Bindy expects to use. They're 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 where each row is separated by a \n(newline) |
| unmarshal | <code>InputStream</code> | <code>List<Map<String, Object>></code> | Contains a List of rows where each row contains 1..n data models contained in a Map |

The important thing to notice in table 3.5 is that Bindy uses a `Map<String, Object>` to represent a CSV row. At first, this may seem odd. Why doesn't it just 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:

- Map key (`String`)—Must contain the fully qualified class name of the model object
- Map value (`Object`)—Must contain the model object

If this seems a bit confusing, don't worry. The following example should make it clearer.

Listing 3.8 Using Bindy to transform a model object to CSV format

```
public class PurchaseOrderBindyTest extends TestCase {
    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,49.95,1\n");
        PurchaseOrder order = new PurchaseOrder(); ①
        order.setAmount(1);
        order.setPrice(new BigDecimal("49.95"));
        order.setName("Camel in Action");
        ProducerTemplate template = context.createProducerTemplate();
②        template.sendBody("direct:toCsv", order);
        mock.assertIsSatisfied();
    }
    public RouteBuilder createRoute() {
        return new RouteBuilder() {
            public void configure() throws Exception {
                from("direct:toCsv")
                    .marshal().bindy(BindyType.Csv,
                                    camelinaction.bindy.PurchaseOrder.class) ③
                    .to("mock:result");
            }
        };
    }
}
```

- ① Creates model object as usual
- ② Starts test
- ③ Transforms model object to CSV

In this listing, you first create and populate the order model using regular Java setters ①. Then you send the order model to the route by sending it to the `direct:toCsv` endpoint ② that is used in the route. The route will then marshal the order model to CSV using Bindy ③. Notice how Bindy is configured to use CSV mode via `BindyType.Csv`. To let Bindy know how to map to order model object, you need to provide a class that is annotated with Bindy annotations.

NOTE Listing 3.8 uses `MockEndpoint` to easily test that the CSV record is as expected. Chapter 9 will cover testing with Camel, and you'll learn all about using `MockEndpoints`.

You can try this example from the chapter3/order directory 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

JSON (JavaScript Object Notation) is a data-interchange format, and Camel provides three components that support the JSON data format: camel-xstream, camel-gson, and camel-jackson. In this section, we'll focus on camel-jackson because Jackson is a very 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. Listing 3.9 shows how you could 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>
        <from uri="jetty://http://0.0.0.0:8080/order"/>
        <bean ref="orderService" method="lookup"/> ②
        <marshal ref="json"/>
    </route>
</camelContext>
```

- ① Sets up JSON data format
- ② Invokes bean to retrieve data for reply

First you need to set up the JSON data format and specify that the Jackson library should be used ①. 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 ② and the `lookup` method is invoked on that bean. The result of this bean invocation is then marshaled to JSON format and returned back 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 section 4.4.5, 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": 49.95}
```

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 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, such as 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. This listing shows how you can configure the CSV data format.

Listing 3.10 Configuring the CSV data format

```
public void configure() {
    CsvDataFormat myCsv = new CsvDataFormat()
        .setDelimiter(';')
        .setHeader(new String[]{"id", "customerId", "date", "item", "amount", "description"}); ①

    from("direct:toCsv")
        .marshal(myCsv) ②
        .to("file://acme/outbox/csv");
}
```

- ① Creates and configures a custom CSV data format
- ② Uses CSV data format

Configuring data formats in Camel is typically done directly on the `DataFormat` itself; sometimes you may also need to use the API that the 3rd party library under the hood

provides. In listing 3.10, the CSV data format nicely wraps the 3rd party API so we can just configure the DataFormat directly ① . Here we set the semicolon as a delimiter and specify the order of the fields ② . The use of the data format stays the same, so all you need to do is refer to it from the `marshal` ③ or `unmarshal` methods. This same principle applies to all data formats in Camel.

Now that you know how to use data formats, let's look at how you can write your own data format.

3.4.6 Writing your own data format

You may find yourself needing to transform data to and from a custom data format. In this section, we'll look at how you can develop a data format that can reverse strings.

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 you could implement a string-reversing data format.

Listing 3.11 Developing a custom data format that can reverse strings

```
package camelinaction;
import java.io.InputStream;
import java.io.OutputStream;
import org.apache.camel.Exchange;
import org.apache.camel.spi.DataFormat;
public class ReverseDataFormat implements DataFormat {
    public void marshal(Exchange exchange, ①
        Object graph, OutputStream stream) throws Exception {
        byte[] bytes = exchange.getContext().getTypeConverter()
            .mandatoryConvertTo(byte[].class, graph);
        String body = reverseBytes(bytes);
        stream.write(body.getBytes());
    }
    public Object unmarshal(Exchange exchange, ②
        InputStream stream) throws Exception {
        byte[] bytes = exchange.getContext().getTypeConverter()
            .mandatoryConvertTo(byte[].class, stream);
        String body = reverseBytes(bytes);
        return body;
    }
    private static 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();
    }
}
```

① Marshals to reverse string

② Unmarshals to unreverse string

The custom data format must implement the `DataFormat` interface, which forces you to develop two methods: `marshal` ① and `unmarshal` ②. That's no surprise, as they're the same methods you use in the route. The `marshal` method ① needs to output the result to the `OutputStream`. To do this, you need to get the message payload as a `byte[]`, and then reverse it with a helper method. Then you write that data to the `OutputStream`. Note that you use the Camel type converters to return the message payload as a `byte[]`. This is very 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. We'll cover Camel's type converters in section 3.6.

To use this new data format in a route, all you have to do is define it as a Spring bean and refer to it 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 this looks like:

```
from("direct:marshal")
    .marshal().custom("reverse")
    .to("log:marshal");
from("direct:unmarshal")
    .unmarshal().custom("reverse")
    .to("log:unmarshal");
```

You'll find this example in the `chapter3/order` directory, and you can try it by using the following Maven goal:

```
mvn test -Dtest=ReverseDataFormatTest
```

You've now learned all about data formats and even how to develop your own. It's time to say goodbye to data formats and take a look at how you can use 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 different template languages:

- *Apache Velocity*—Probably the best known templating language (<http://camel.apache.org/velocity.html>)
- *FreeMarker*—Another popular templating language that may be a bit more advanced than Velocity (<http://camel.apache.org/freemarker.html>)

These two templating languages are fairly similar to use, so we'll only discuss 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.
```

There are three pieces of information in the email that must be replaced at runtime with real values. What you need to do is 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 we've inserted \${ } placeholders in the template, which instructs Velocity to evaluate and replace them at runtime. Camel prepopulates the Velocity context with a number of entities that are then available to Velocity. Those entities are listed in table 3.6.

NOTE The entities in table 3.6 also apply for other templating languages, such as FreeMarker.

Table 3.6 Entities prepopulated in the Velocity context and that are 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 <code>request</code> 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 <code>response</code> 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 leverage 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
```

TIP For more details on the Camel Velocity component, consult the online documentation (<http://camel.apache.org/velocity.html>).

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 About 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, we'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 you can 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 the `TypeConverterRegistry` and the `TypeConverters` it holds.

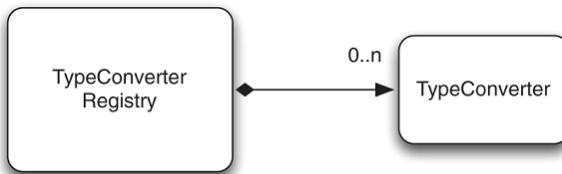


Figure 3.7 The `TypeConverterRegistry` contains many `TypeConverters`

The `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 the `TypeConverter`, Camel can then convert one type to another 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 not only type converters 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.

To scan and load the type converters, Camel uses `org.apache.camel.impl.converter.AnnotationTypeConverterLoader`. 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 or not 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.CxfPayloadConverter
```

The `AnnotationTypeConverterLoader` will load those classes that have been annotated with `@Converter`, and then it searches within them for public methods that are annotated with `@Converter`. Each of those methods is considered a type converter. A package name of `org.apache.camel.component.cxf.converter` could have also been provided here but Camel will load them more quickly when they are specified directly.

This process is best illustrated with an example. The following code is a snippet from `IOConverter` class from 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 using `javax.jmx.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("activemq:queue:inbox");
```

If you're using Spring XML, you provide the type as an attribute instead, like this:

```
<route>
<from uri="file://riders/inbox"/>
<convertBodyTo type="java.lang.String"/>
<to uri="activemq:queue:inbox"/>
</route>
```

You can omit the `java.lang.` prefix on the type, which can shorten the syntax a bit:
`<convertBodyTo type="String"/>.`

Another reason for using `convertBodyTo` is to read files 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("activemq:queue:inbox");
```

TIP 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 cannot 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 you can 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 we looked at how type converters work.

Suppose you wanted 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 a `@Converter` class containing the type-converter method.

Listing 3.12 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
        String s = converter.convertTo(String.class, data);
        if (s == null || s.length() < 30) {
            throw new IllegalArgumentException("data is invalid");
        }
        s = s.replaceAll("#START##", ""); ②
        s = s.replaceAll("#AND##", ""); 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
            .convertTo(Integer.class, s3); ②
        return new PurchaseOrder(name, price, amount); ②
    } ①
}

```

- ① Grabs TypeConverter to reuse
- ② Converts from String to PurchaseOrder

In listing 3.12, 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 how you can use the `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 magic 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 using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderConverterTest
```

RETURNING NULL VALUES

By default a null return value from a type converter is not valid. Camel considers null as a "miss" and adds the pair of types you are 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 3.12 required a null return value you could do something like:

```
@Converter(allowNull = true)
public static PurchaseOrder toPurchaseOrder(byte[] data,
                                             Exchange exchange) {
    ...
}
```

ADDING TYPE CONVERTERS TO CAMEL-CORE

If you are 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 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 is just a dummy package name,

the actual 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 different 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 using EIPs and beans. You also learned that Camel provides special support for transforming XML documents 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. We'll cover how to use beans in chapter 4.

In the preceding two chapters, we've covered two crucial features of integration kits: routing and transformation. The next chapter dives into the world of beans, and you'll see how Camel can easily adapt to and leverage 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
- 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 different component models, such as CORBA, EJB, JBI, SCA, and lately OSGi. Some of these models, 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 models. This left the everyday engineer 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 POJO model.

In the wake of this many open source projects has proven that the POJO programming model and a lightweight container indeed meet the expectations of today's businesses. In fact, the simple programming model and lightweight container concept proved superior to the heavyweight and over-complex enterprise application and integration servers that were used before. This trend continues this day and we are seeing the raise of micro services and container less deployments. We will cover all about micro services with Camel in chapter 7, at

first we need to learn more basics with Camel such as this chapter about using Java beans with Camel.

So what does this have to do with 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 such as the Spring Framework, with both of them being lightweight containers favoring the POJO programming model.

In fact, Camel recognizes the power of the POJO programming model and goes great lengths to work with your beans. By using beans, you fulfill an important goal in the software industry, which is to reduce 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.

We'll start this chapter by showing you how *not* to use beans with Camel, which will make it clearer how you should use beans. After that, we'll take a look at the theory behind the Service Activator EIP and dive inside Camel to see how this pattern is implemented. We then spend much time covering 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. We end the chapter with covering how you can use beans as predicates and expressions in your route.

4.1 Using beans the hard way and the easy way

In this section, we'll walk through an example that shows how not to use beans with Camel—the hard way to use beans. Then we'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 some different ways you could 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.

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
                public void process(Exchange exchange) throws Exception {

```

```

        String name = exchange.getIn().getBody(String.class);
        HelloBean hello = new HelloBean();          ②
        String answer = hello.hello(name);          ②
        exchange.getOut().setBody(answer);
    }
});
}

```

- ① Uses a processor**
② Invokes HelloBean

Listing 4.1 shows a `RouteBuilder`, which defines the route. You use an inlined Camel Processor ①, which gives you the `process` method, in which you can work on the message with Java code. First, you must extract the message body from the input message, which is the parameter you'll use when you invoke the bean later. Then you need to instantiate the bean and invoke it ②. Finally you must set the output from the bean on the output message.

Now that you've done it the hard way using the Java DSL, let's take a look at using Spring XML.

4.1.2 Invoking a bean defined in XML DSL

When using Spring or OSGi Blueprint as a bean container then 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 the HelloBean

```

<bean id="helloBean" class="camelinaction.HelloBean"/>           ①
<bean id="route" class="camelinaction.InvokeWithProcessorSpringRoute"/>

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

```

- ① Defines HelloBean**

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 a `CamelContext`, which is the way you get Spring and Camel to work together.

Now let's take a closer look at the route.

Listing 4.3 A Camel route using a Processor to invoke HelloBean

```

public class InvokeWithProcessorSpringRoute extends RouteBuilder {           ①
    @Autowired
    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);
            exchange.getOut().setBody(answer);
        }
    });
}

```

- ① Injects HelloBean
- ② Invokes HelloBean

The route in listing 4.3 is nearly identical to the route in listing 4.1. The difference is that now the bean is 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 there's a bit of plumbing to get it all to work. Here are some reasons why it's hard to work with beans:

- 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

Suppose you were to define the Camel route in the Spring XML file instead of using a `RouteBuilder` class. The following snippet shows how this might be done:

```

<bean id="helloBean" class="camelaction.HelloBean"/>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        < What goes here >           ①
        </route>
    </camelContext>

```

- ① Insert something here to use beans

First you define the bean as a Spring bean, and then you define the Camel route with the `direct:start` input. At ① you want to invoke `HelloBean`, but you're in trouble—this is XML, and you can't add Java code in the XML file.

In Camel, the easy way to use beans is to use the `<bean>` tag at ① :

```
<bean ref="helloBean" method="hello"/>
```

That gives you the following route:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <bean ref="helloBean" method="hello"/>
  </route>
</camelContext>
```

Camel offers the same solution when using the Java DSL. You can simplify the route in listing 4.3 like this:

```
public void configure() throws Exception {
    from("direct:hello").beanRef("helloBean", "hello");
}
```

That's a staggering reduction from eight lines of code to one. 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`.

You could even omit the `hello` method, because the bean only has a single method:

```
public void configure() throws Exception {
    from("direct:hello").beanRef("helloBean");
}
```

Using the `<bean>` tag is an elegant solution for working with beans. Without using that tag, you had to use a Camel `Processor` to invoke the bean, which is a tedious solution.

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 simply as

```
from("direct:hello").bean(HelloBean.class);
```

Now let's look at how you can work with beans in Camel from the EIP perspective.

4.2 The Service Activator pattern

The Service Activator pattern is an enterprise pattern described in Hohpe and Woolf's *Enterprise Integration Patterns* book (<http://www.enterpriseintegrationpatterns.com/>). It

describes a service that can be invoked easily from both messaging and non-messaging services. Figure 4.1 illustrates this principle.

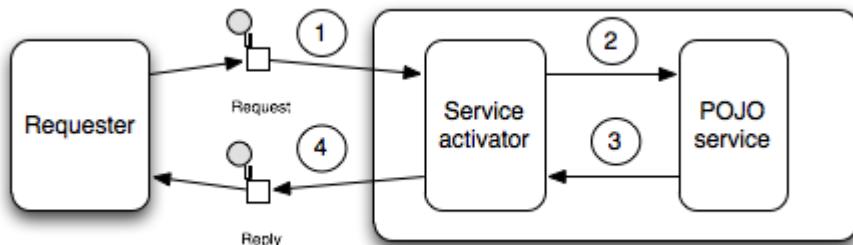


Figure 4.1 The service activator mediates between the requestor and the POJO service.

Figure 4.1 shows a service activator component that 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 the `org.apache.camel.component.bean.BeanProcessor` to do the work. We'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 we looked at in section 4.1.3, as illustrated in figure 4.2.

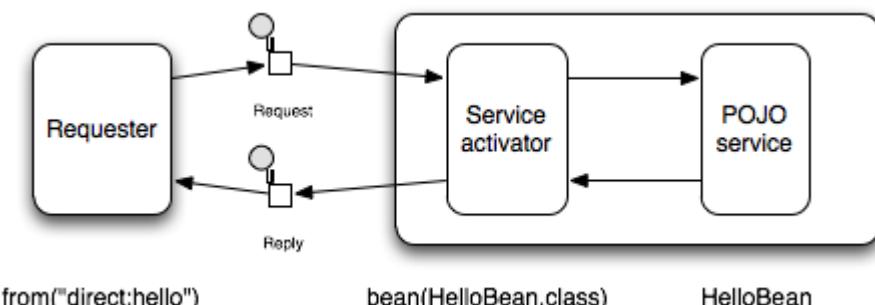


Figure 4.2 Relationship between a Camel route and the Service Activator EIP

Figure 4.2 shows 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 different registries.

4.3 Camel's bean registries

When Camel works with beans, it looks them up in a registry to locate them. Camel's philosophy is to leverage 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.

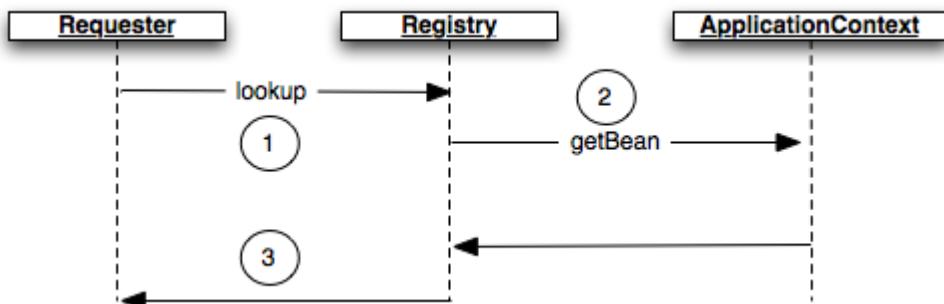


Figure 4.3 A requester looks up a bean using the Camel registry, which then uses the Spring ApplicationContext to determine where the bean resides.

Figure 4.3 shows that the Camel registry is an abstraction that sits between the caller and the real registry. When a 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. 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> findByNameWithClass(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 the `HelloBean`, you would 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 typesafe 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 `findByTypeWith Name` and `findByType`, is mostly used internally by Camel to support convention over configuration—it allows 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

| Registry | Description |
|----------------------------|--|
| JndiRegistry | An implementation that uses an existing Java Naming and Directory Interface (JNDI) registry to look up beans. |
| SimpleRegistry | An in-memory only registry which uses a <code>java.util.Map</code> to hold the entries. |
| ApplicationContextRegistry | An implementation that works with Spring to look up beans in the Spring <code>ApplicationContext</code> . 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 lookup beans from the OSGi service registry and as well in the Blueprint Container. This implementation is automatically used when you're using Camel in a OSGi Blueprint environment. |
| CdiBeanRegistry | An implementation that works with CDI to lookup beans in the CDI Container. This implementation is used when using the <code>camel-cdi</code> component. |

In the following sections, we'll go over each of these six registries.

4.3.1 JndiRegistry

The `JndiRegistry`, as its name implies, 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();
```

The `JndiRegistry` (like the `SimpleRegistry`) is often used for testing or when running Camel standalone. Many of the unit tests in Camel use the `JndiRegistry` because they were created before the `SimpleRegistry` was added to Camel.

NOTE The `JndiRegistry` is being considered to be replaced with `SimpleRegistry` as the default registry in Camel 3.0 onwards.

The source code for the book contains an example of using the `JndiRegistry` which is identical to the next example using `SimpleRegistry` shown in listing 4.4. We recommend to 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

The `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 the `SimpleRegistry` to enlist the `HelloBean` and refer to it from the route.

Listing 4.4 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(); ①
        registry.put("helloBean", new HelloBean()); ①

        context = new DefaultCamelContext(registry); ②

        template = context.createProducerTemplate();
        context.addRoutes(new RouteBuilder() {
            public void configure() throws Exception {
                from("direct:hello").beanRef("helloBean");
            }
        });
        context.start();
    }
}
```

```

    }

    protected void tearDown() throws Exception {
        template.stop();                                ③
        context.stop();
    }

    public void testHello() throws Exception {
        Object reply = template.requestBody("direct:hello", "World");
        assertEquals("Hello World", reply);
    }
}

```

- ① Registers HelloBean in SimpleRegistry
- ② Uses SimpleRegistry with Camel
- ③ Cleans up resources after test

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 a `ProducerTemplate`, which makes it simple to send messages to Camel, as can be seen in the test method. Finally, when the test is done, you clean up the resources by stopping Camel ③ . In the route, you use the `beanRef` 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

The `ApplicationContextRegistry` is the default registry when Camel is used with Spring. More precisely, it's the default when you set up Camel in the Spring XML, as this snippet illustrates:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <bean ref="helloBean" method="hello"/>
    </route>
</camelContext>
```

Defining Camel using the `<camelContext>` tag will automatically let Camel use the `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 applies 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 will look up whether a service with the name exists in the OSGi service registry. If not, Camel will fall back and look up the name in the regular registry, such as the Spring-DM ApplicationContextRegistry, or BlueprintContainer when using OSGi Blueprint.

Popular OSGi platforms with Camel

The most popular OSGi platforms to use with Apache Camel is Apache Karaf, Apache ServiceMix. You can also find commercial platforms from vendors such as Red Hat and Talend. Red Hat offers the popular fuse brand that, at the time of writing, is available as JBoss Fuse (based on OSGi with Karaf) and Fuse EAP (based on JavaEE with WildFly).

Suppose you want to allow other bundles in OSGi to reuse the following bean:

```
<bean id="helloBean" class="camelinaction.HelloBean"/>
```

This can be done by exporting the bean as an OSGi service which will enlist the bean into the OSGi Service Registry. This is done as follows:

```
<osgi:service id="helloService" interface="camelinaction.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="camelinaction.HelloBean"/>
```

To call the bean from the Camel route is easy, as its just using the bean component as if the bean is a local `<bean>` element in the same Spring XML file.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
<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 Spring bean container for you. This is convention over configuration.

The last registry is when you use Camel with CDI.

4.3.5 CdiBeanRegistry

CDI (Context and Dependency Injection) 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 is part of JavaEE, and therefore available in JEE application servers such as Apache TomEE, WildFly, and in commercial offerings as well. However CDI is lightweight and you can run in a standalone CDI container such as Apache OpenWebBeans or JBoss Weld.

We will encounter CDI again in chapter 7 where the topic is microservices.

The `CdiBeanRegistry` is the default registry when using Camel with CDI. The registry is automatically created in the default constructor of the `CdiCamelContext` as shown below:

```
public class CdiCamelContext extends DefaultCamelContext {

    public CdiCamelContext() {
        super(new CdiBeanRegistry());
        setInjector(new CdiInjector(getInjector()));
    }
}
```

①

① using `CdiBeanRegistry` as bean registry when using Camel with CDI

This means as a Camel end user you do not need to configure this, as Camel with CDI is automatically configured.

In CDI beans can be defined using CDI annotations. For example to define a singleton bean with the name `helloBean` you could do as shown in listing 4.5

Listing 4.5 - A simple bean to print hello world using CDI annotations

```
import javax.inject.Named;
import javax.inject.Singleton;

@Singleton
@Named("helloBean")
public class HelloBean {
    private int counter;

    public String hello() {
        return "Hello " + ++counter + " times";
    }
}
```

①

②

- ① - The bean is singleton scoped
- ② - The bean is registered with the name `helloBean`

To use the bean from a Camel route we can use a bean reference to lookup the bean by its name ② . The Camel route could also be coded with CDI as shown in listing 4.6

Listing 4.6 - A Camel route using CDI to use the bean from listing 4.5

```
@ContextName("myCamel")
public class HelloRoute extends RouteBuilder {  
    ①  
  
    @EndpointInject(uri = "timer:foo?period=5s")
    private Endpoint input; ②  
  
    @EndpointInject(uri = "log:output")
    private Endpoint output; ③  
  
    @Override
    public void configure() throws Exception {
        from(input)
            .beanRef("helloBean") ④
            .to(output);
    }
}
```

- ① - This route is added to the CamelContext with name myCamel
- ② - Inject Camel endpoint which are used in the route
- ③ - Inject Camel endpoint which are used in the route
- ④ - Invoke the bean by referring to the bean name

The route builder class has been annotated with `org.apache.camel.cdi.ContextName` ① which is used to add the route to the CamelContext with the provided name. Then we inject endpoints using `org.apache.camel.EndpointInject` ② ③ . This is not needed as we could have used the endpoint uris directly in the Java DSL route, which could have been written as follows:

```
from("timer:foo?period=5s")
    .beanRef("helloBean")
    .to("log:output");
```

However we have chosen to show a practice often used with CDI which is dependency injection, and why we inject the endpoints also. To call the bean we can use `.beanRef` ④ to refer to the bean by its name. However we could also have injected the bean using CDI. This can be done using `@Inject` and `@Named` as shown in the code snippet below:

```
@Inject @Named("helloBean")
private Object helloBean;  
  
public void configure() throws Exception {
    from(input)
        .bean(helloBean)
        .to(output);
}
```

NOTE We will cover a lot more about using CDI with Camel in the future such as chapters 9 and 15.

This example is provided in the source code of the book in chapter4/cdi-beans. You can try the example with the following Maven goal:

```
mvn clean install exec:java
```

This concludes our 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 actual 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 leverages the registry to look up the bean to invoke.

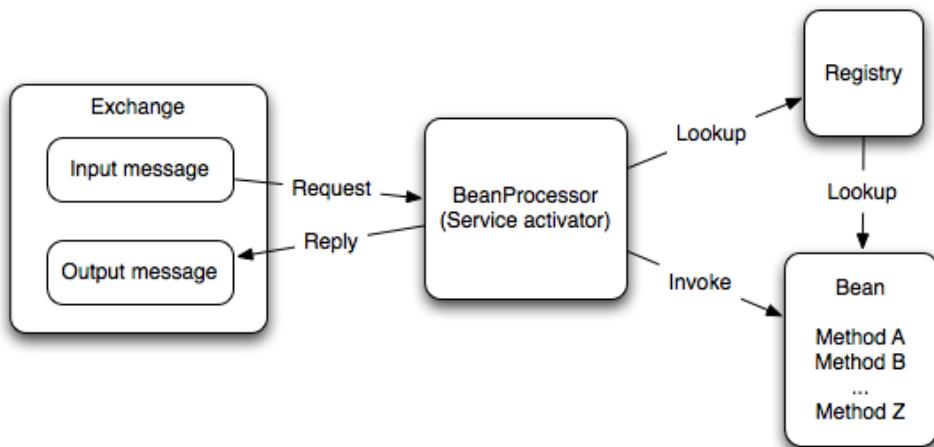


Figure 4.4 To invoke a bean in Camel, the bean component (BeanProcessor) looks it up in the registry, selects and adapts a method, invokes it, and passes the returned value as the reply to the Camel exchange.

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

We've covered how registry lookups are done in section 4.2. The next two steps (steps 2 and 3 in the preceding list) are more complex, and we'll cover them in the remainder of this chapter. The reason why this is more complex in Camel is because 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);
```

There are a total of three methods for Camel to select among. 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 will ensure 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 had misspelled the method name, perhaps typing "ekko", the compiler could not 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` does not 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 or not, 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 won'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, and it's continued in figure 4.6.

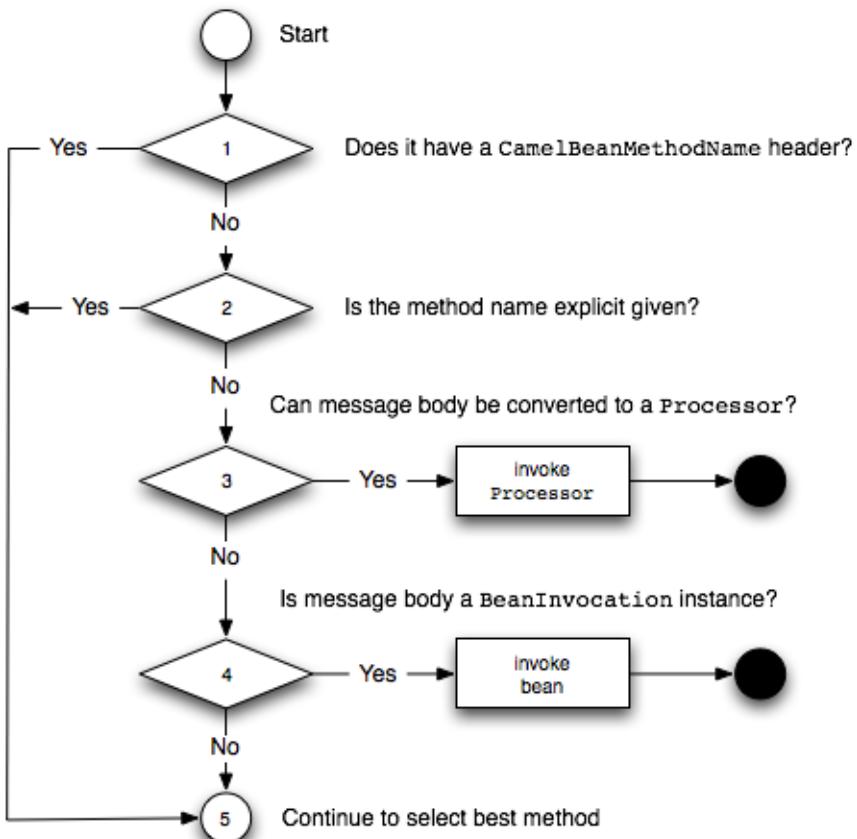


Figure 4.5 How Camel selects which method to invoke (part 1, continued in figure 4.6)

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 a bit 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 an `org.apache.camel.component.bean.BeanInvocation`, that's used to invoke the method and pass the arguments to the bean. This is only in use when using Camel bean proxy which we will cover in chapter 7.
5. Continued in the second part of the algorithm, shown in figure 4.6.

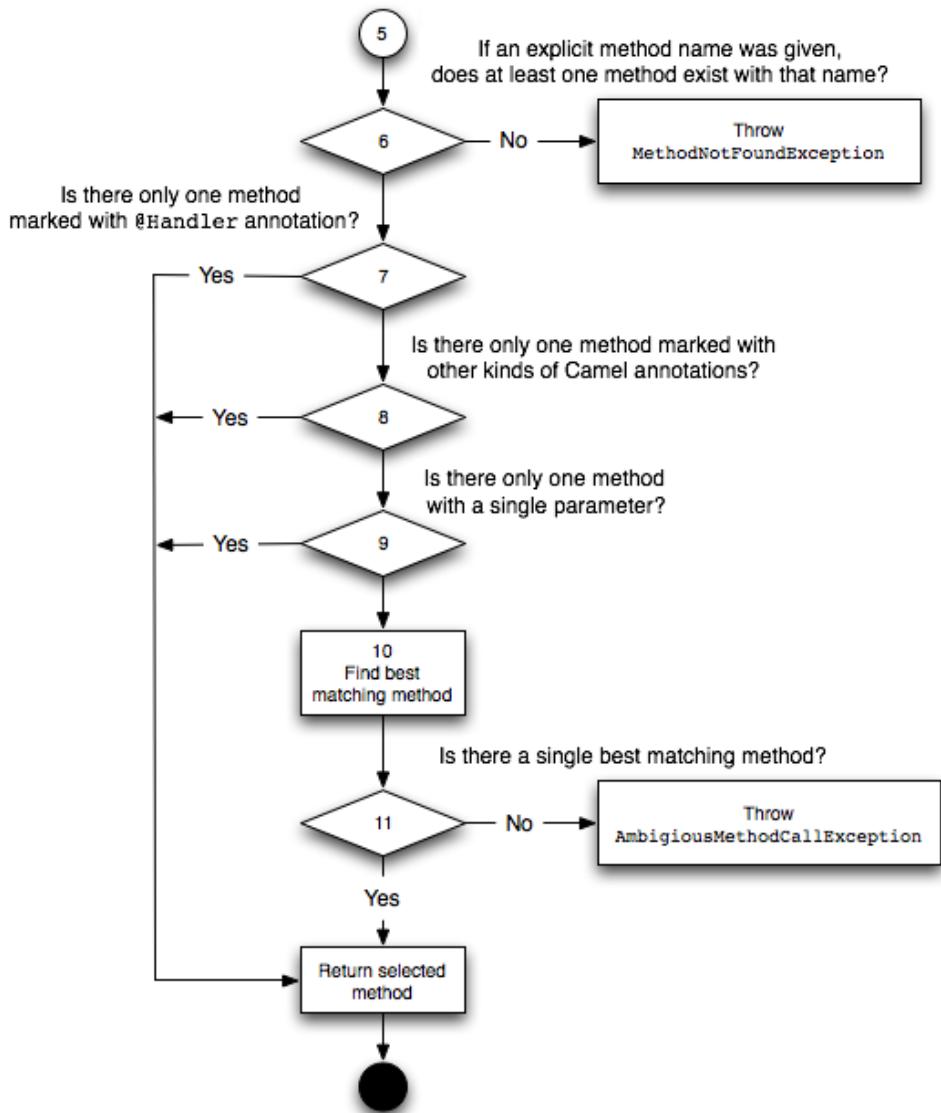


Figure 4.6 How Camel selects which method to invoke (part 2, continued from figure 4.5)

Figure 4.6 is a bit more complex, but its main goal is to narrow down the number of possible methods and select a method if one stands out. Don't worry if you don't entirely understand the algorithm; we'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. (We'll look at how Camel binds to method parameters using annotations in section 4.3.3.)
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 we looked at in section 4.3.1, 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 there's a single method that stands out as the best fit. The strategy is to go over the candidate methods and filters 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 was given including any number of parameter values, then these values are also used during filtering. The values are matched with the `parring` parameter type on the candidate methods. In the end, if there is only a single method left, that method is selected. Since this logic is elaborate we will cover this in more details in the following sections.
6. If Camel can't select a method, an `AmbigiousMethodCallException` 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 2.16. This method-selection algorithm may change in the future to accommodate new features.

Now it's time to take a look at how this algorithm applies in practice.

4.4.3 Some method-selection examples

To see how this algorithm works, we'll use the `EchoBean` from section 4.3.1 as an example, but we'll add another method to it—the `bar` method—to better explain what happens when there are multiple candidate methods.

```
public class EchoBean {

    public String echo(String echo) {
        return echo + " " + eco;
    }

    public String bar() {
        return "bar";
    }
}
```

And we'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. Despite the fact that `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 is only one of them.

To make the example a bit 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 now 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 `AmbigiousMethodCallException` exception, according to step 11 in figure 4.6.

How can you resolve this? One solution would be 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 dealt with in step 7 of figure 4.6. The `@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 the `AmbigiousMethodCallException` 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 double(Integer num) {
        return num.intValue() * num.intValue();
    }
}
```

The `echo` method works with a `String`, and the `double` 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 is an exact type match in any of the methods.

Suppose you send in 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 a `String`. On the other hand, if you send in a message with the `Integer` value of 5, Camel will select the `double` method, because it uses the `Integer` type.

Despite this, things can still go wrong, so let's go over a couple of common situations.

4.4.4 Potential method-selection problems

There are a few things that 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 only happens when you have 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 was 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:

```
.beanRef("echoBean", "foo")
```

And in XML DSL:

```
<bean ref="echoBean" method="foo"/>
```

Here you try to invoke the `foo` method, but there is no such method, so Camel throws a `MethodNotFoundException` exception.

On the other hand, you could omit specifying the method name:

```
.beanRef("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`:

```
public class OrderServiceBean {
    public String handleXML(Document xml) {
        ...
    }
}
```

And suppose you need to use that bean in this route:

```
from("jms:queue:orders")
    .beanRef("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 stracktrace:

```
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.createTypeConversionException
n(BaseTypeConverterRegistry.java:571)
at
org.apache.camel.impl.converter.BaseTypeConverterRegistry.convertTo(BaseTypeConverterR
egistry.java:129)
at
org.apache.camel.impl.converter.BaseTypeConverterRegistry.convertTo(BaseTypeConverterR
egistry.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)

```

What happened is that Camel tried to convert the `javax.jms.TextMessage` to a `org.w3c.dom.Document` type, but it failed. In this situation, Camel wraps the error and throws it as a `TypeConversionException` exception.

By further looking into this stacktrace, 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 is not XML based.

NOTE You may wonder what would happen if such a situation occurred at runtime. 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 there is one last functionality we would like you to know about. It covers use-cases where your beans have overloaded methods (aka. methods using the same name, but with different parameter types etc.) and how Camel works in those situations.

4.4.5 Method selection using type matching

The algorithm we previously talked about in step 10 from 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")
    .beanRef("orderService", "handleXML")
    .to("jms:queue:handledOrders");
```

The `OrderServiceBean` have two methods named `handleXML` each one accepting a different parameter type as the input. What would happen at runtime depends on whether the bean component is able to find a single suitable method to invoke accordingly to the algorithm listed in figure 4.5 and 4.6. To decide that Camel will 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 then there is a direct match with the types on those methods and the appropriate method is invoked. However if the Camel message body is of any other type, such as `java.io.File`, then Camel will lookup whether there is type converters that can convert from `java.io.File` and respectively to `org.w3c.Document` and `java.lang.String`. If there is only one type conversion possible then the fitting method is chosen. In any other case an `AmbigiousMethodCallException` is throw.

In those situations you can assist Camel by explicit defining which of the two methods to call by specifying which type to use, as shown below:

```
from("jms:queue:orders")
    .beanRef("orderService", "handleXML(org.w3c.Document)")
    .to("jms:queue:handledOrders");
```

You would need to specify the class type using its fully qualified name. However for common types like `Boolean`, `Integer`, `String` etc, you can omit the package name, and use just `String` as the `java.lang.String` type:

```
from("jms:queue:orders")
    .beanRef("orderService", "handleXML(String)")
    .to("jms:queue:handledOrders");
```

Over the years with Camel we do not see the need for this very often. If possible we suggest to use unique methods 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 we need to look at the bean parameter-binding process, which happens after Camel has selected the method.

4.5 Bean parameter binding

In the last section, we 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*.

We've already seen parameter binding in action in the many examples so far in this chapter. What those examples had in common was using a single parameter to which Camel bound the input message body. Figure 4.7 illustrates this using EchoBean as an example.

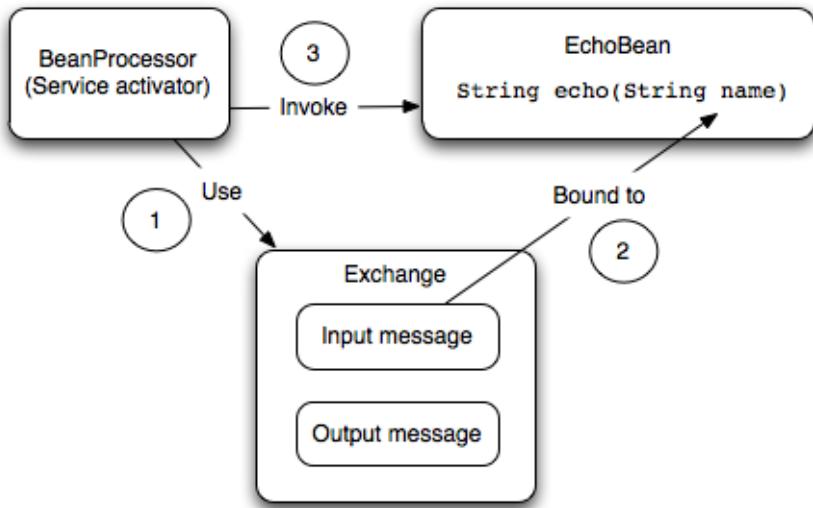


Figure 4.7 How bean component binds the input message to the first parameter of the method being invoked

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.

So 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.

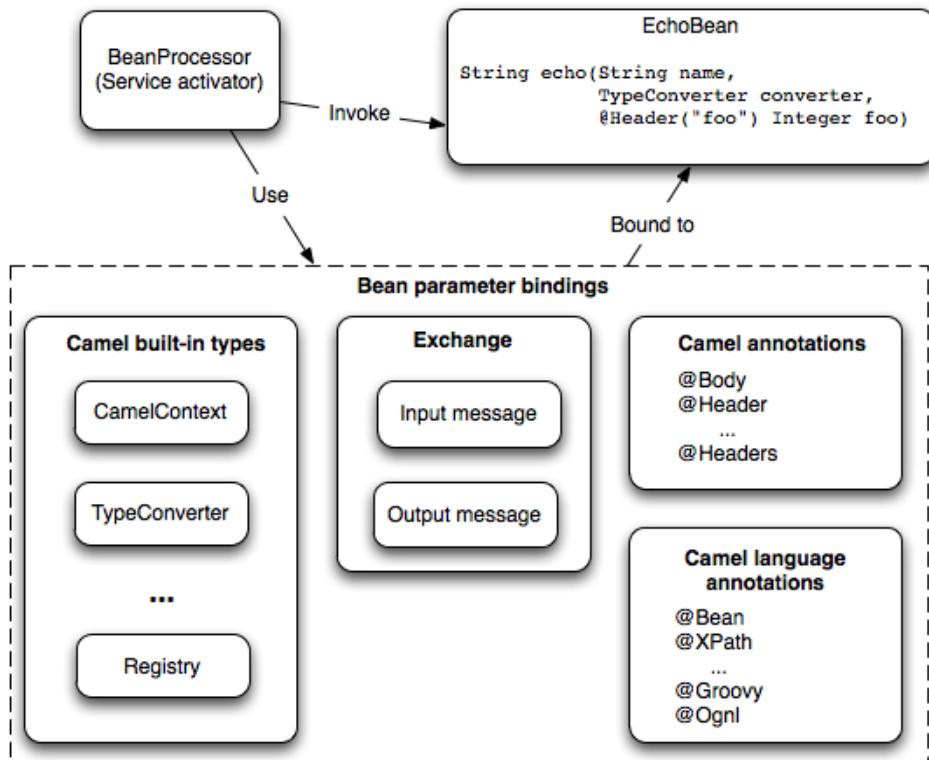


Figure 4.8 Parameter binding with multiple parameters involves a lot more options than with single parameters.

At first, figure 4.8 may seem a bit 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’ll cover them in section 4.4.2.
- *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 will be covered in the sections to come.
- *Camel annotations*—When dealing with multiple parameters, you use annotations to distinguish them. This is covered in section 4.4.3.
- *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 4.4.4.

In addition you can specify the parameter binding using the method name signature which resemble how you call methods in Java source code. We will take a look at this in section 4.4.5.

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 as 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 specify the binding in the method name signature which makes it clear for humans and Camel how each parameter should be mapped. We will cover this in section 4.4.5.

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 |
|---------------|--|
| Exchange | The Camel exchange. This contains the values that will be bound to the method parameters. |
| Message | The Camel input message. It contains the body that is often bound to the first method parameter. |
| CamelContext | The CamelContext. This can be used in special circumstances when you need access to all Camel's moving parts. |
| TypeConverter | The Camel type-converter mechanism. This can be used when you need to convert types. We covered the type-converter mechanism in section 3.6. |
| Registry | The bean registry. This allows you to look up beans in the registry. |
| Exception | An exception, if one was thrown. Camel will only bind to this if the exchange has failed and contains an exception. This allows you to use beans to handle errors. |

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 the `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 will explain 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 how 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

| Annotation | Description |
|---------------------------|--|
| <code>@Attachments</code> | Binds the parameter to the message attachments. The parameter must be a <code>java.util.Map</code> type. |

| | |
|-------------------------|---|
| @Body | Binds the parameter to the message body. |
| @Header(name) | Binds the parameter to the given message header. |
| @Headers | Binds the parameter to all the input headers. The parameter must be a java.util.Map type. |
| @ExchangeProperty(name) | Binds the parameter to the given exchange property. |
| @Properties | Binds the parameter to all the exchange properties. The parameter must be a java.util.Map type. |
| @ExchangeException | Binds the parameter to an Exception set on the exchange. |

You've already seen the first two types in action, so let's try a couple of 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:

```
public String orderStatus(@Body Integer orderId,
    @Header("customerId") Integer customerId,
    @Header("customerType") Integer customerType) {
    ...
}
```

If you have many headers, then 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 language 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 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 language annotations provided in Camel 2.16. In the future, we may add additional languages to Camel, which often also means that a corresponding annotation for bean parameter binding is added as well.

Table 4.4 Camel's language-based bean binding annotations

| Annotation | Description | Maven Dependency |
|--------------------------|---|------------------|
| <code>@Bean</code> | Invokes a method on a bean | camel-core |
| <code>@Constant</code> | Evaluates as a constance value | camel-core |
| <code>@EL</code> | Evaluates an EL script (unified JSP and JSF scripts) | camel-juel |
| <code>@Groovy</code> | Evaluates a Groovy script | camel-script |
| <code>@JavaScript</code> | Evaluates a JavaScript script | camel-script |
| <code>@JsonPath</code> | Evaluates a JSONPath expression | camel-jsonpath |
| <code>@JXPath</code> | Evaluates a JXPath expression | camel-jxpath |
| <code>@MVEL</code> | Evaluates a MVEL script | camel-mvel |
| <code>@OGNL</code> | Evaluates an OGNL script | camel-ognl |
| <code>@PHP</code> | Evaluates a PHP script | camel-script |
| <code>@Python</code> | Evaluates a Python script | camel-script |
| <code>@Ruby</code> | Evaluates a Ruby script | camel-script |
| <code>@Simple</code> | Evaluates a Simple expression (Simple is a built-in language provided with Camel; see appendix A for more details) | camel-core |
| <code>@SpEL</code> | Evaluates a Spring Expression | camel-spring |
| <code>@SQL</code> | Evaluates a JO-SQL expression | camel-josql |
| <code>@Terser</code> | Evaluates a HL7 terser expression | camel-hl7 |
| <code>@XPath</code> | Evaluates an XPath expression | camel-core |
| <code>@XQuery</code> | Evaluates an XQuery expression | camel-saxon |

Most popular languages

Table 4.4 list many different languages you can use. It is our experience that most people only use a few of these languages such as @Bean, @Groovy, @JsonPath, @Simple, @SpEL, @XPath or @XQuery.

The most commonly used goes without saying is @Simple that can easily access any parts of the message body and headers. For evaluating Java code @Groovy is popular, and @SpEL by users whom are familiar with the Spring Expression Language. If you work with JSON or XML structures then @JsonPath, @XPath, and @XQuery are good choices. To call a method on a Java bean you can use @Bean which we cover right now.

It may seem a bit magical that you can use a @Bean annotation when invoking a method, because the @Bean annotation itself also invokes a method. Let's try out an example.

Suppose you already have a service that must be used to stamp unique order IDs on incoming orders. The service is implemented as follows.

Listing 4.7 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");
    attr.setValue("") + orderId);
    Node node = xml.getElementsByTagName("order").item(0);
    node.getAttributes().setNamedItem(attr);
    return xml;
}
```

- ① Creates orderId attribute
- ② Adds orderId attribute to order node

As you can see, the service creates a new XML attribute with the value of the given order ID ①. Then it inserts this attribute in the XML document ② using the rather clumsy XML API from Java ②.

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="camelinaaction.XmlOrderService"/>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file://riderautoparts/order/inbox"/>
        <bean ref="xmlOrderService"/>
    </route>
</camelContext>
```

```

<to uri="jms:queue:order"/>
</route>
</camelContext>
```

What is missing is the step that generates a unique ID and provides that ID in the `handleIncomingOrder` method (shown in listing 4.7). To do this, you need to declare a bean in the spring XML file with the ID generator, as follows:

```
<bean id="guid" class="camelinaaction.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 4.7. To do this, you use the `@Bean` annotation and change the method signature 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:

```
2015-05-17 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>]
2015-05-17 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 did not 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 using the `@NamespacePrefix` annotation embedded in the `@XPath` annotation. Notice the XPath expression value must use the prefix, which means the expression is changed from `/order/@customerId` to `/c:order/@customerId`.

In recent time JSON has become increasing more popular to use as data format when exchanging data. Now imagine if the previous example is using JSON instead of XML, then lets see how the Camel `@JsonPath` binding annotation works in practice.

USING `@JSONPATH BINDING ANNOTATION`

To use `@JsonPath` we 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>2.16.0</version>
</dependency>
```

Instead of dealing with incoming orders as XML documents, the orders are now in JSON format as a sample shown below:

```
{
  "order": {
    "customerId": 4444,
    "item": "Camel in Action"
  }
}
```

Listing 4.8 shows how we transform incoming orders in JSON format to a CSV representation ② by mapping the `customerId` and `item` fields ① to bean parameters using the `@JsonPath` annotation.

Listing 4.8 A Service that transforms JSON to CSV using a Java bean

```
import org.apache.camel.jsonpath.JsonPath;
import org.apache.camel.language.Bean;

public class JsonOrderService {

    public String handleIncomingOrder(
        @JsonPath("$.order.customerId") int customerId,          ①
        @JsonPath("$.order.item") String item,
        @Bean(ref = "guid", method = "generate") int orderId) {

        return String.format("%s,%s,%s", orderId, customerId, item); ②
    }
}
```

① bindings from JSON document to bean parameters using `@JsonPath`

② returns a CSV representation of the incoming data

The source code for the book contains this example in the chapter4/json directory. Maven users can run the example using the following Maven goal:

```
mvn test -Dtest=JsonOrderTest
```

When it's running, you should see a log that shows after the transformation from the service bean, such as:

```
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 expression and predicates. For more information about the syntax consult the JsonPath documentation at <https://github.com/jayway/JsonPath>

What we have seen in this and the previous section is the need for using Camel annotation to declare the needed binding information. This is common practice to use Java annotations, but the caveat is that it requires you to alter the source code of the bean to add these annotations. 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 to specify the parameter binding information using a syntax that is very 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 very powerful technique as it completely decouples Camel from your Java bean. In other words your Java bean can stay *as-is* without having to import any Camel code or dependencies at compile nor at run time. The caveat is that the binding must be defined in a String value which 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 we covered in the previous section is not available in the Simple language.

It's actually easier to explain with an example.

Suppose we have this method we used previously as an example in section 4.4.3:

```
public String orderStatus(@Body Integer orderId,
                           @Header("customerId") Integer customerId,
                           @Header("customerType") Integer customerType) {
    ...
}
```

Now instead of using the Camel annotations the source code becomes:

```
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, that mean the code can compile without having Camel JARs on the classpath. So what we have to do is to specify the binding details in the Camel route instead

```
from("direct:start")
    .beanRef("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>
```

So what happened? If you take a closer look you will see that the method name parameter almost resemble Java source code, as if calling a method with 3 parameters. If you were to write some Java code and use the Camel Exchange API to call the method from Java, then the source code would be something alike:

```
OrderStatus bean = ...
Message msg = exchange.getIn();
String status = bean.orderStatus(msg.getBody(), msg.getHeader("customerId"),
    msg.getHeader("customerType"));
```

The code above would be actual Java source code and therefore is compiled by the Java compiler. So the parameter binding happens on the compile time. Where as the preceding Camel routes are defined using a String value in Java code, or an XML attribute. Since its not the Java compiler that performs the binding, its Camel parsing the String value; which happens on starting up Camel. Figure 4.9 illustrates how each of the three Java parameters corresponds to a value.

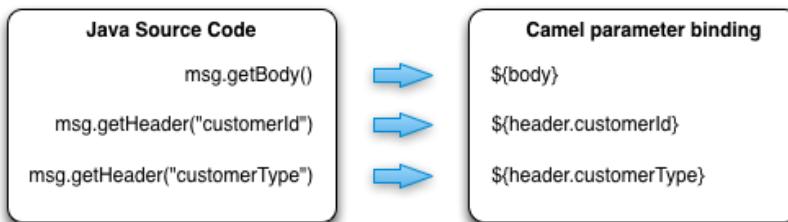


Figure 4.9 - How bean parameter bindings in Camel resemble Java code when calling a parameter with 3 values.

You may have guessed what syntax the Camel parameter binding is using, from the right hand side in figure 4.9? Yes its the simple language. Camel allows you to specify the bean parameter binding using the method name signature. Table 4.5 lists the rules that apply when using parameter binding using the method name signature.

Table 4.5 - Listing the rules the bean parameter binding using method name signature supports.

| Rule | Description |
|---------------------|---|
| A boolean value | The parameter can bind to a boolean type by using the value as either true or false. For example <code>method="goldCustomer(true)".</code> |
| A numeric value | The parameter can bind to a number type by using the value as an integer. For example <code>method="goldCustomer(true, 123)".</code> |
| A null value | The parameter can bind as a null value. For example <code>method="goldCustomer(true, 123, null)".</code> |
| A literal value | The parameter can bind to a String type by using the value as a literal. For example <code>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 cannot apply to any of the above 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 (table 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 then 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 we saw in chapter 3, to perform message translation using Java code. On top of that Camel also makes it very easy to use beans as decision makers during routing.

4.6 Using beans as predicates and expressions

Some enterprise integration patterns (EIPs) uses predicates to determine how they should process messages such as the content based router, and message filter. And other EIPs requires using expressions, such as the recipient list, dynamic router, idempotent consumer and others. This section covers how we can use Java beans as predicates or expressions with those EIPs.

First lets briefly remind ourselves what a Camel predicate and expression is.

A predicate is an expression that evaluates as a `boolean`; i.e. 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; i.e. 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 we have learned from chapter 2, one of the most commonly used EIP pattern is the content based router. This pattern uses one or more predicates to determine how messages are routed. If a predicate matches then the message is routed down the given path.

The example we will use is a customer order system that routes orders from gold, silver and regular customers. A bean is used to determine which kind of customer level the routed message is from. A very simple implementation is shown in listing 4.9.

Listing 4.9 - Using a bean with methods to be used as predicates in Camel

```
public class CustomerService {

    public boolean isGold(@JsonPath("$.order.customerId") int customerId) {
        return customerId < 1000; ①
    }

    public boolean isSilver(@JsonPath("$.order.customerId") int customerId) {
        return customerId < 5000; ②
    }
}
```

① method to determine if its a gold customer

② method to determine if its a silver customer

The bean implements two methods `isGold` ① and `isSilver` ② both returning a `boolean` which allows Camel to leverage the methods as predicates. Each method uses bean parameter binding to map the from the Camel message to the customer id. In this example we continue from the previously JSON example and use the `@JsonPath` annotation to extract the customer id from the message body in JSON format.

TIP The bean in listing 4.9 is using the bean parameter binding which we covered in section 4.7. Therefore you can use what you have learned and instead of @JsonPath the bean could have multiple parameters and what else we learned about in section 4.7.

To use the bean as predicate in the content based router in Camel is easy as showing in listing 4.10.

Listing 4.10 - The content based router uses the bean as predicate using XML

```
<bean id="customerService" class="camelinaction.CustomerService"/> ①

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://target/order"/>
    <choice>
      <when>
        <method ref="customerService" method="isGold"/> ②
        <to uri="mock:queue:gold"/>
      </when>
      <when>
        <method ref="customerService" method="isSilver"/> ③
        <to uri="mock:queue:silver"/>
      </when>
      <otherwise>
        <to uri="mock:queue:regular"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

- ① Declare the bean so we can refer to the bean in the route below
- ② The method call predicate which invokes the isGold method on the bean
- ③ The method call predicate which invokes the isSilver method on the bean

As you can see from listing 4.10, using a bean as predicate is easy, by using `<method>` ② ③ to setup the bean as the predicate. Prior to that the bean needs to be declared using a `<bean>` element ① .

Listing 4.11 shows how to implement the same route as in listing 4.10 but using Java DSL.

Listing 4.11 - The content based router using the bean as predicate using Java DSL

```
public void configure() throws Exception {
  from("file://target/order")
    .choice()
      .when(method(CustomerService.class, "isGold")) ①
        .to("mock:queue:gold")
      .when(method(CustomerService.class, "isSilver")) ②
        .to("mock:queue:silver")
      .otherwise().to("mock:queue:regular");
}
```

- ① The method call predicate which invokes the isGold method on the bean
- ② The method call predicate which invokes the isSilver method on the bean

The route in listing 4.11 is almost identical to listing 4.10. However this time we show a slight variation in ① ② which allows you to refer to the bean by its classname, `CustomerService.class`. When doing this then Camel may be required to create a new instance of the bean when the route is being created, and therefore the bean is required to have a default no-argument constructor. However if the method `isGold` or `isSilver` is static methods, then Camel will not create a new bean instance, but invoke the static methods directly.

The Java DSL could also like in listing 4.10 refer to the bean by a id, which would require to pass in the name instead of the class name as shown in the snippet below:

```
.when(method("customerService", "isGold"))
.to("mock:queue:gold")
```

In Java DSL there is more as you can combine multiple predicates into compound predicates.

USING COMPOUND PREDICATES IN JAVA

This is a exclusive feature in Java only, that allows you to combine one ore more predicates into a compound predicate. This can be used to logically combine (and, or, not) multiple predicates together, even if they use different languages. For example combing XPath, Simple and Bean as shown in listing 4.12.

Listing 4.12 - 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(
            xpath("//book/title = 'Camel in Action'"),
            simple("${header.source} == 'batch'"),
            not(method(CompoundPredicateTest.class, "isAuthor")));
        from("direct:start")
            .validate(valid)
            .to("mock:valid");
    }
};
```

- ① Use `PredicateBuilder` to combine the predicates using `and` (all must return true)
- ② The XPath predicate which tests if the book title is `Camel in Action`
- ③ The simple predicate which tests a header.`source` has the value `batch`
- ④ The method call predicate calls the `isAuthor` method which is negated using `not`, and therefore should return `false`
- ⑤ Use the compound predicate in the Camel route using the validate EIP pattern

The meat is the `org.apache.camel.builder.PredicateBuilder` ① which has a number of builder methods to combine predicates. We use `and` which means all the three ② ③ ④ predicates must return true for the compound predicate to return true. If one of them returns false, then 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 expression in routes which is the next topic.

4.6.2 Using beans as expressions in routes

In this section we will cover a common use-case with Camel which is how to route using a *-dynamic to*. In other words how can you route a message in Camel to a destination which is dynamically computed at runtime with information from the message itself.

In the enterprise integration patterns book, it would be the recipient list pattern that best describe the *dynamic to*, that is illustrated in figure 4.10.

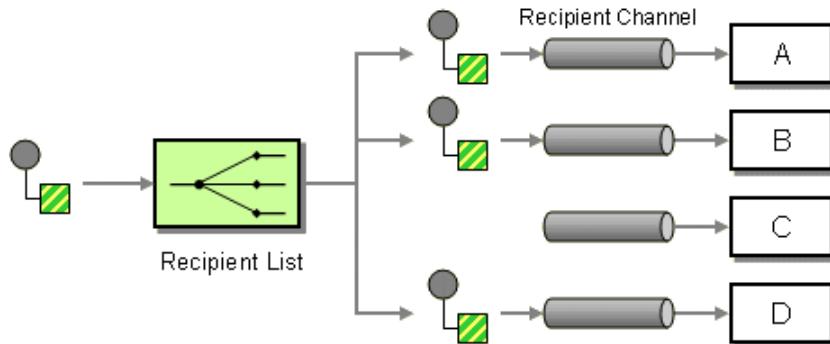


Figure 4.10 - Recipient List EIP pattern sends a copy of the same message to a number of dynamic computed destinations. The *dynamic to* is merely just a single dynamic computed destination.

The recipient list EIP pattern in Camel is a very versatile and flexible implementation, which offers many features and functions. This EIP is covered in more details in the following chapter.

To use a bean with the recipient list, we use a simple use-case with a customer service system that route orders depending on geographical region of the customer. A naive bean could be implemented in a few lines of code as shown below:

```
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";
        }
    }
}
```

To use the bean as expression in the recipient list based router in Camel is also very easy as showing in listing 4.13.

Listing 4.13 - Using a bean as expression during routing with the recipient list to act as a dynamic to

```
<bean id="customerService" class="camelinaction.CustomerService"/> ①

<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
        </setHeader>
        <recipientList>
            <simple>mock:queue:${header.region}</simple> ③
        </recipientList>
    </route>
</camelContext>
```

- ① Bean to be used as expression implementing logic to determine customer geographical location
- ② Calling the bean to set a header with the region of the customer
- ③ Dynamic to using recipient list to route to a queue with the name of the region

First we need to setup the bean as a `<bean>` ① so we can refer to the bean using its id - `customerService`. Then the bean is invoked to return from which geographical region the customer order is placed. 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 its also referred as *dynamic to*. If the customer is from US, then the destination would be "mock:queue:US", and for EMEA it would be "mock:queue:EMEA" etc.

Listing 4.13 could be implemented in Java DSL with a fewer lines of code as shown below:

```
from("file:///target/order")
    .setHeader("region", method(CustomerService.class, "region"))
    .recipientList(simple("mock:queue:${header.region}"));
```

The source code of this book includes this example in chapter4/expression directory, which you can run using the following Maven goals:

```
mvn test -Dtest=JsonExpressionTest
mvn test -Dtest=SpringJsonExpressionTest
```

In the example we didn't call the bean from the recipient list, but instead computed the region as a header. We 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 you will do:

```
<from uri="file://target/order"/>
<recipientList>
    <simple>mock:queue:${bean:camelinaction.CustomerService?method=region}</simple>
</recipientList>
```

That is all we had to say about using beans as predicates and expression. In fact we have 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 leverage 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 you can 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 expressions to parameters, which is a great feature when working with XML 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 leverage 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.
- *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 this out itself. 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 its often easier to specify the parameter binding in the method name signature in the route which closely resemble 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 in need for defining predicates or expressions using a more powerful language then 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.

Camel also allows to use beans for routing (aka POJO routing) which users enjoy using when developing Camel applications in a micro service style. We will cover this in chapter 7.

Back in chapter 2, you were exposed to some of Camel's routing capabilities by using some standard EIPs. In the next chapter, we'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 is 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 the complexity of integration problems, 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: <http://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 around 60 patterns. This chapter is devoted to covering five of the most powerful and feature-rich patterns. The patterns discussed in this chapter are 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, where 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 using a variety of different balancing policies. This pattern is covered in section 5.6. |

Let's look at these patterns in a bit 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 out 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.

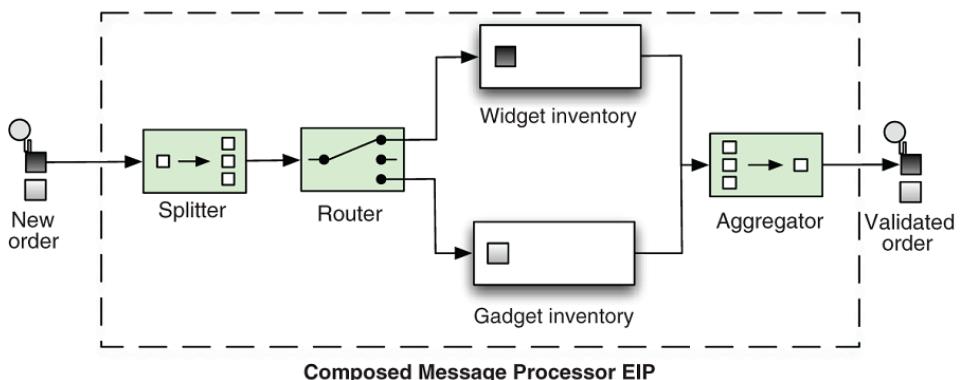


Figure 5.1 The Composed Message Processor EIP splits up the message, routes the submessages to the appropriate destinations, and re-aggregates the response back into a single message.

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 is often asked on the Camel mailing list is how to route messages dynamically. The answer is to use EIPs such as Recipient List, Routing Slip, and Dynamic Router. We covered Recipient List in chapter 2, and in this chapter we'll 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 we'll cover 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 well. Don't despair if you don't understand the pattern in the first few pages.

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.

Once a completion condition occurs, the aggregated message is sent to the output channel for further processing. We'll cover how this process works in detail in the next section.

Example uses of Aggregator

The Aggregator EIP supports many use cases, such as the loan broker example from the EIP book, where 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 5 seconds. This can be done using the Aggregator to choose the latest message and thus trigger a completion every 5 seconds.

When using the Aggregator, you have to pay attention to the following three configuration settings, which must be configured. Failure to do so will cause Camel to fail on startup and to 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, we'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 Introducing the Aggregator EIP

Suppose you want to collect any three messages together and combine them together. 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 would work. 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 when three messages have been aggregated, 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:

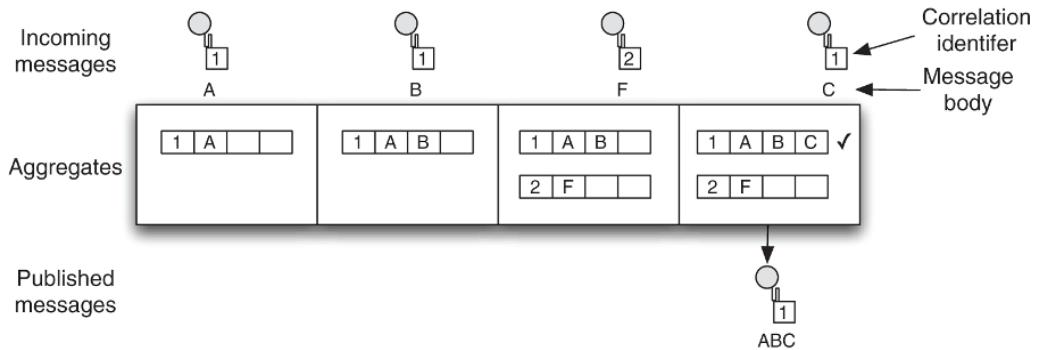


Figure 5.3 Illustrates the Aggregator EIP in action, with partial aggregated messages updated with arriving messages.

As mentioned before, there are three configurations 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 simply states that when three messages have been aggregated, the completion should trigger.

The same example in Spring 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>

```

```
</route>
</camelContext>
```

The Spring XML is a little different than the Java DSL because you define the AggregationStrategy using the strategyRef attribute on the <aggregate> tag. This refers to a Spring <bean>, which is listed in the top of the Spring XML file. The completion condition is also defined as a completionSize attribute. The most noticeable difference is how the correlation identifier is defined. In Spring XML, it is defined using the <correlationExpression> tag, which has a child tag that includes the Expression.

The source code for the book contains this example in the chapter5/aggregator directory. You can run the examples 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 will 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 how 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 has not been met, so it waits in the aggregator. You could modify the test method to send in 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.aggregation` package, and it defines a single method:

```
public interface AggregationStrategy {
    Exchange aggregate(Exchange oldExchange, Exchange newExchange);
}
```

If you are having a *déjà vu* moment, its most likely because `AggregationStrategy` is also used by the `Content Enricher` EIP, which we covered in TODO chapter 3.

Listing 5.1 shows the strategy used in the previous example.

Listing 5.1 AggregationStrategy for merging messages together

```
import org.apache.camel.Exchange;
import org.apache.camel.processor.aggregate.AggregationStrategy;
public class MyAggregationStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        if (oldExchange == null) {                                         ①
            return newExchange;
        }
        String oldBody = oldExchange.getIn()                                ②
            .getBody(String.class);
        String newBody = newExchange.getIn()                                ②
            .getBody(String.class);
        String body = oldBody + newBody;                                     ②
        oldExchange.getIn().setBody(body);
        return oldExchange;
    }
}
```

- ① Occurs for a new group
- ② Combines message bodies

At runtime, the `aggregate` method is invoked every time a new message arrives. In this example, it will be invoked four times: one for each arriving message *A*, *B*, *F*, and *C*. To show how this works, we've listed the invocations as they would happen, in table 5.2.

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 messages 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 simply want to return the message as is, because there are no other messages to combine it with ① .

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 ② . Then you replace the existing body in the `oldExchange` with the updated body.

NOTE The Aggregator EIP uses synchronization, which ensures that the `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 are 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'll discuss this and review the different 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 where 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 timeout condition that would react if all messages aren't received within a certain time period.

To cater for that situation and others, Camel provides seven different completion conditions, which are listed in table 5.3. You can mix and match them according to your needs.

Table 5.3 Different kinds of completion conditions provided by the Aggregator EIP

| Condition | Description |
|--------------------------------|---|
| <code>completionSize</code> | 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 <code>Expression</code> to dynamically decide a size at runtime. |
| <code>completionTimeout</code> | Defines a completion condition based on an inactivity timeout. This condition triggers if a correlation group has been inactive longer than the specified period. Timeouts are scheduled for each correlation group, so the timeout is individual to each group. You can either use a fixed value (<code>long</code>) or use an <code>Expression</code> to |

| | |
|------------------------------|---|
| | dynamically decide a timeout at runtime. The period is defined in milliseconds. You can't use this condition together with the completionInterval. |
| completionInterval | Defines a completion condition based on a scheduled interval. This condition triggers periodically. There is a single scheduled timeout for all correlation groups, which causes all groups to complete at the same time. The period (<code>long</code>) is defined in milliseconds. You can't use this condition together with the <code>completionTimeout</code> . |
| completionPredicate | Defines a completion condition based on whether the <code>Predicate</code> matched. See also the <code>eagerCheckCompletion</code> option in table 5.5. This condition is enabled automatically if the <code>aggregationStrategy</code> implements either <code>PredicateOrPreCompletionAwareAggregationStrategy</code> . In the case of <code>PreCompletionAwareAggregationStrategy</code> this gives you the ability to complete the aggregation group on receipt of a new Exchange and start a new group with the new Exchange. |
| completionFromBatchConsumer | Defines a completion condition that is only applicable when the arriving Exchanges are coming from a <code>BatchConsumer</code> (http://camel.apache.org/batch-consumer.html). A number of components support this condition, such as: Atom, File, FTP, Hbase, Mail, MyBatis, Jclouds, SNMP, SQL, SQS, S3, and JPA. |
| forceCompletionOnStop | Defines a completion condition that will complete all correlation groups on shutdown of the CamelContext. |
| Using an AggregateController | By using an <code>AggregateController</code> you can control group completion externally via Java calls into the controller or via JMX. |

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 source code for the book 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 you can use multiple completion conditions.

USING MULTIPLE COMPLETION CONDITIONS

The source code for this book contains an example in the chapter5/aggregator directory showing how to use multiple completion conditions. You can run the example 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 an additional completion condition.

The same example in Spring 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 will 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 timeout, 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. However, 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 timeout condition ensures that after a period of inactivity the message will be published. In that regard, you can use the timeout condition as a fallback condition, with the price being that the published message will only be partly aggregated. Suppose you expected two messages to be aggregated into one, but you only received 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> | Integer | The total number of arrived messages aggregated. |
| <code>Exchange.AGGREGATED_COMPL</code> <code>ETED_BY</code> | String | <p>The condition that triggered the completion. Possible values are "size", "timeout", "interval", "predicate", "force", "strategy", and "consumer".</p> <p>The "consumer" value represents the completion from batch consumer.</p> |

| | | |
|--|----------------------------------|--|
| Exchange.AGGREGATED_CORRELATION_KEY | String | The correlation identifier as a String. |
| Exchange.AGGREGATED_TIMEOUT | Long | The timeout in milliseconds as set by the completion timeout. |
| Exchange.AGGREGATION_STATEGY | Map<Object, AggregationStrategy> | A map of AggregationStrategy objects with a MulticastProcessor as the key. Used when aggregating replies from a multicast. |
| Exchange.AGGREGATION_COMPLETE_CURRENT_GROUP | boolean | Set this to true to complete the current group. |
| Exchange.AGGREGATION_COMPLETE_ALL_GROUPS | boolean | Set this to true to complete all groups and ignore the current Exchange. |
| Exchange.AGGREGATION_COMPLETE_ALL_GROUPS_INCLUSIVE | String | 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 to the console which condition completed, simply 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 or not 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 timeout occurred, and not all expected message 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 |
|----------------------|---------|--|
| eagerCheckCompletion | false | This option specifies whether or not to eager-check for completion. Eager checking means Camel will check for completion conditions before aggregating. By default, Camel will check for completion after aggregation. |

| | | |
|---|-------|---|
| | | This option is used to control how the completionPredicate condition behaves. If the option is false, the completion predicate will use the aggregated Exchange for evaluation. If true, the incoming Exchange is used for evaluation. |
| closeCorrelationKey -OnCompletion | null | 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 ClosedCorrelationKeyException is thrown. |
| | | This option uses an Integer parameter that represents a maximum bound for a least recently used (LRU) cache, which keeps track of closed correlation keys. Note that this cache is in-memory only and will be reset if Camel is restarted. |
| ignoreInvalid- CorrelationKeys | false | This option specifies whether or not to ignore invalid correlation keys. By default, Camel throws a CamelExchangeException for invalid keys. You can suppress this by setting this option to true, in which case Camel skips the invalid message. |
| groupExchanges | false | This option is used for grouping arriving Exchanges into a single combined Exchange holder that contains the Exchanges. If it's enabled, you should not configure an AggregationStrategy. |
| timeoutCheckerExecu torService / timeoutCheckerExecu torServiceRef | null | 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. |
| optimisticLockRetry Policy | null | Specifies a 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 source code for the book in the chapter5/aggregator directory. You can run test examples 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 how you can implement aggregation strategies without using any Camel API at all.

USING POJOS FOR THE AGGREGATIONSTRATEGY

So far we've seen that you can customize how Exchanges are aggregated by implementing AggregationStrategy in a custom class. There is a slightly cleaner way of doing this however, without using Camel APIs at all. Like Camel's bean integration, you can provide the aggregator with a POJO to act as the AggregationStrategy. Camel handles injecting one or all of the message body, headers and Exchange properties. Taking a look at the AggregationStrategy used in Listing 5.1, we 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 is much cleaner than the AggregationStrategy in listing 5.1. Similarly though, it has 2 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 like:

```
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 additional parameters in pairs and also order is important – the first parameter is the body, second headers, and third properties.

The Camel route looks a bit different from before as well. When referencing the above AggregationStrategy POJO your route looks like:

```
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 we used the `AggregationStrategies` utility class to convert the POJO into a `AggregationStrategy`. There are many more methods available in this class as well. For instance, you can pass in a bean reference, select the method you want to use, or even just the class.

The same route in Spring XML is shown here:

```
<bean id="myAggregationStrategy" class="camelinaction.MyAggregationStrategyPojo"/>

<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 out for yourself there are examples in the source code for the book in the chapter5/aggregator directory. You can run test examples 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 like: `JdbcAggregationRepository`, `CassandraAggregationRepository`, `LevelDBAggregationRepository`, and `HazelcastAggregationRepository`. We'll 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>.

We'll look at how you can 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 Spring XML you would 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 here.

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");
```

Here's the same example in Spring XML.

Listing 5.3 Using LevelDB with Aggregator in Spring XML

```
<bean id="myAggregationStrategy"
    class="camelinaction.MyAggregationStrategy"/>
<bean id="myRepo"
    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">
            <correlationExpression>
                <constant>true</constant>
            </correlationExpression>
            <log message="Sending out ${body}"/>
        </aggregate>
    </route>
</camelContext>
```

① LevelDB persistent repository

As you can see from listing 5.3, 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 using the `aggregationRepositoryRef` attribute on the `aggregate` tag.

RUNNING THE EXAMPLE

The source code for the book contains this example in the chapter5/aggregator directory. You can run it 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 will display 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 example, 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:

```
2015-05-17 12:18:24,846 [ #1 - file://target/inbox] INFO route1 - Consuming file a.txt
2015-05-17 12:18:29,383 [ #1 - file://target/inbox] INFO route1 - Consuming file b.txt
...
2015-05-17 12:20:08,052 [ main] INFO DefaultCamelContext - Apache Camel 2.15.0
(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:

```
2015-05-17 12:22:23,320 [ main] INFO LevelDBAggregationRepository - On startup there are 1
aggregate exchanges (not completed) in repository: myrepo
2015-05-17 12:22:28,380 [ #1 - file://target/inbox] INFO route1 - Consuming file c.txt
2015-05-17 12:22:28,414 [ #1 - file://target/inbox] INFO route1 - Sending out ABC
```

Notice how it logs on startup how many exchanges are in the persistent repository. In this example there is 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 place where messages may be lost:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

messages that have been published (send out) from the Aggregator, could potentially fail during routing as well.

To remedy this problem you could use one of these two approaches:

- *Camel error handlers* (*covered in TODO chapter 5*)—these provide redelivery and dead letter channel capabilities.
- *A RecoverableAggregationRepository* – `RecoverableAggregationRepository` is an interface extending `AggregationRepository`, which offers the recovery, redelivery, and dead letter channel features. Camel provides four such repositories out of the box: `JdbcAggregationRepository`, `CassandraAggregationRepository`, `LevelDBAggregationRepository`, and `HazelcastAggregationRepository`.

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 only deal with this by retrying or eventually giving up and moving the message to a dead letter channel.

The `RecoverableAggregationRepository` on the other hand, is tightly integrated into the Aggregator, which allows additional benefits such as leveraging the persistence store for recovery and offering transactional capabilities. It ensures 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, we've provided the following 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.

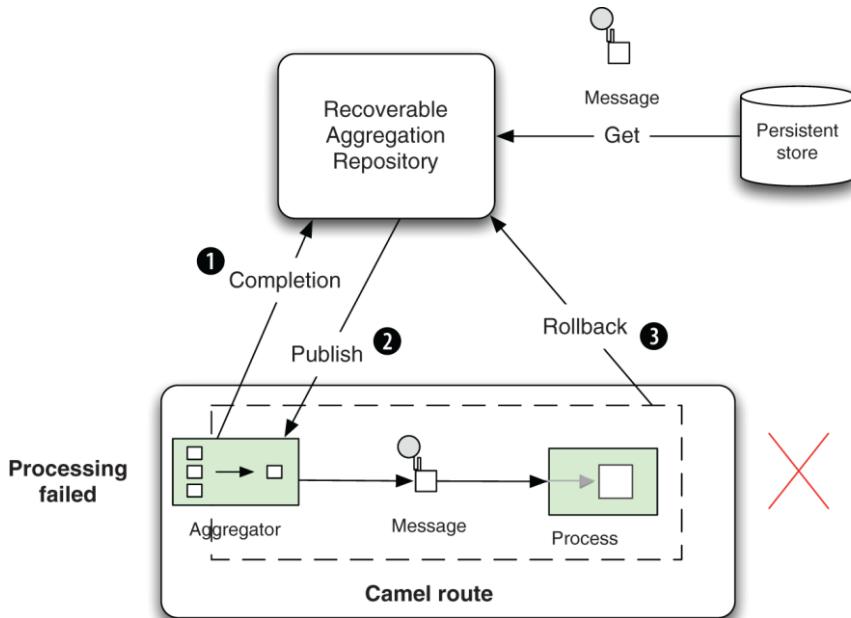


Figure 5.4 An aggregated message is completed ①, it's published from the Aggregator ②, and processing fails ③, so the message is rolled back.

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 the `RecoverableAggregationRepository`, which can act accordingly.

Now imagine the same message is recovered and redelivered, as shown in figure 5.5.

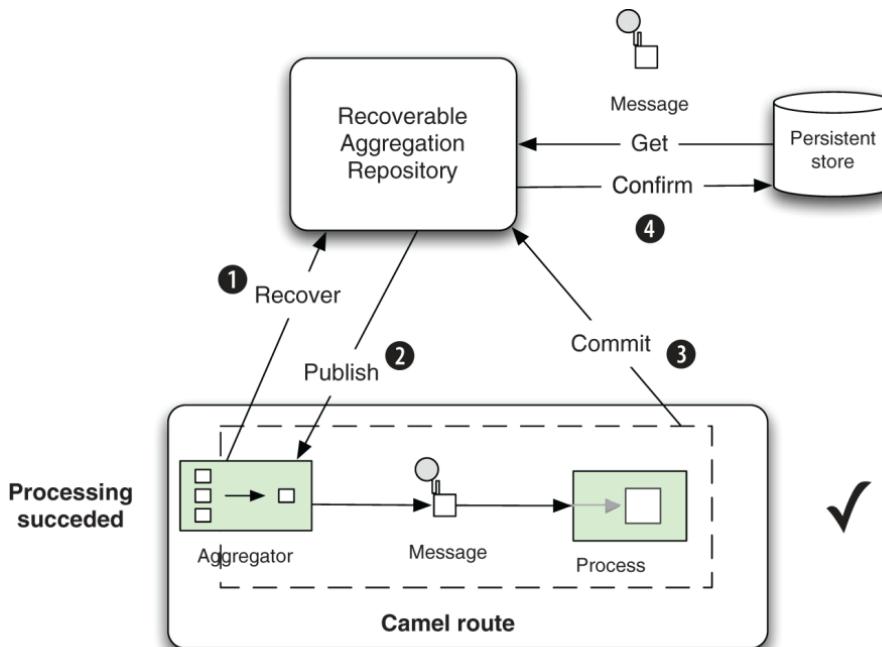


Figure 5.5 The Aggregator recovers ① failed messages, which are published again ② , and this time the messages completed ③ successfully ④ .

The Aggregator uses a background task, which runs every second, 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 ④ the message, ensuring it won't be recovered on subsequent scans.

NOTE The transactional behavior provided by `RecoverableAggregationRepository` isn't based on Spring's `TransactionManager` (which we'll cover in TODO chapter 9). The transactional behavior is based on LevelDB's own transaction mechanism.

RUNNING THE EXAMPLE

The source code for the book contains this example in the `chapter5/aggregator` directory. You can run it 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. This means that 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 Spring XML, you can set this up as a spring `<bean>` tag, as follows:

```
<bean id="myRepo" class="org.apache.camel.component.leveldb.LevelDBAggregationRepository">
<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 we'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. This means that after 4 failed recovery attempts, the message is exhausted and is 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:

```
2015-05-20 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 code 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? Obviously 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 |
|--|------|---|
| <code>Exchange.REDELIVERY_COUNTER</code> | 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.REDELIVERED | 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 only available 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 the RecoverableAggregationRepository that are related to recovery.

Table 5.7 RecoverableAggregationRepository configuration options related to recovery

| Option | Default | Description |
|---------------------|---------|---|
| useRecovery | true | Whether or not recovery is enabled. |
| recoveryInterval | 5000 | How often the recovery background tasks are executed. The value is in milliseconds. |
| deadLetterUri | null | An optional dead letter channel, where published messages that are exhausted should be sent. This is similar to the DeadLetterChannel error handler, which we covered in TODO chapter 5 . This option is disabled by default. When in use, the maximumRedeliveries option must be configured as well. |
| maximumRedeliveries | null | 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.

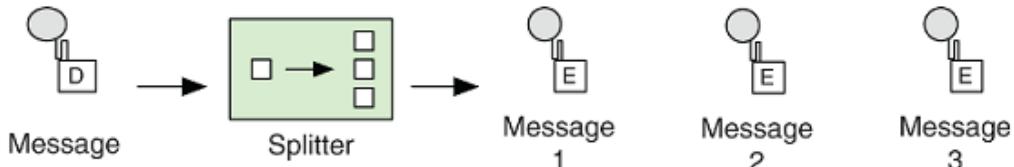


Figure 5.6 The Splitter breaks out the incoming message into a series of individual messages.

In this section, we'll teach you all you need to know about the Splitter. We'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*. Listing 5.4 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()) ①
                    .log("Split line ${body}")
                    .to("mock:split");
            }
        };
    }
}

```

① Splits incoming message body

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 Spring XML, you have to do this a bit differently because the Splitter uses an `Expression` to return what is to be split.

In the Java DSL we defined the `Expression` shown in bold:

```
.split(body())
```

Here, `body()` is a method available on the `RouteBuilder`, which returns an `org.apache.camel.Expression` instance. In Spring 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 Spring XML, you use the Camel's expression language, known as Simple (discussed in appendix A), to tell the Splitter that it should split the message body.

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=SplitterABCTest
mvn test -Dtest=SpringSplitterABCTest
```

Now you've seen the Splitter in action. To better understand how you can 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.

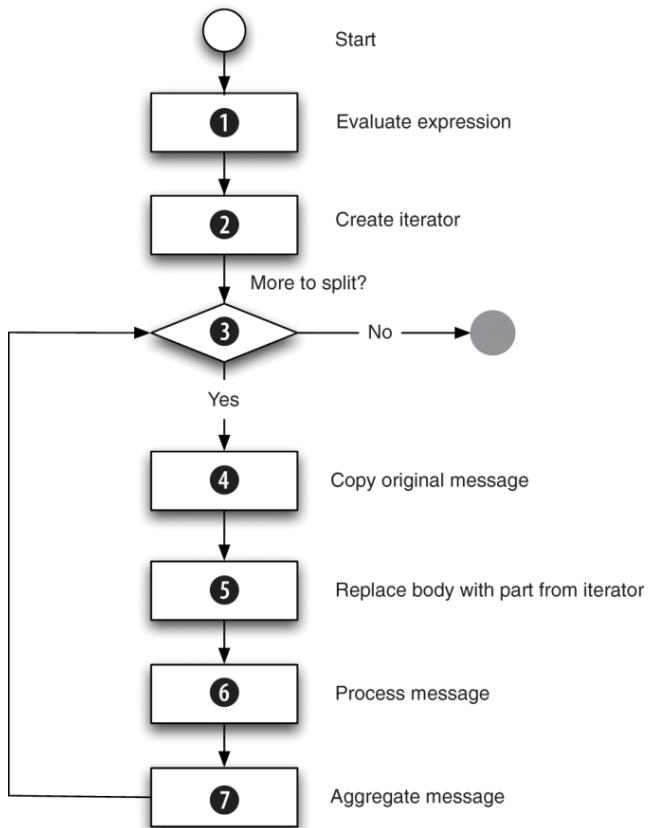


Figure 5.7 A sequence diagram showing how the Splitter works internally, by using an iterator to iterate through the message and process each entry.

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 a `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`, `Array`, `org.w3c.dom.NodeList`, `String` (with entries separated by commas). Any other type will be iterated once.

Then the Splitter uses the iterator ③ until there is no more data. Each message to be sent out of the iterator is a copy of the message ④, which has had its message body replaced 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.SPLIT_INDEX | Integer | The index for the current message being processed. The index is zero-based. |
| Exchange.SPLIT_SIZE | Integer | The total number of messages 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.SPLIT_COMPLETE | Boolean | Whether or not this is the last message being processed. |

You may find yourself in a situation where you need 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. 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:

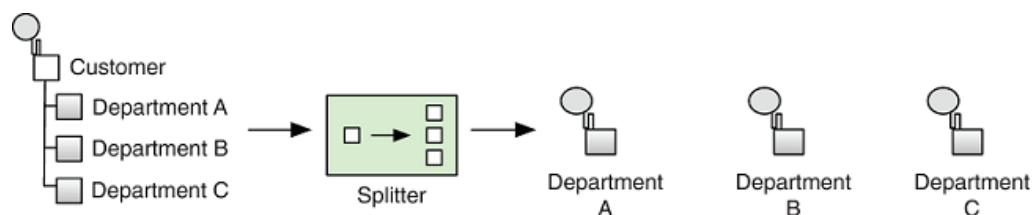


Figure 5.8 Splitting a complex message into submessages by department

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 Spring XML, you'd have to declare the `CustomerService` in a Spring bean tag, as follows:

```
<bean id="customerService" class="camelinaaction.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 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=SplitterBeanTest
mvn test -Dtest=SpringSplitterBeanTest
```

The logic in the `splitDepartments` method is simple, but it shows how you can 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("${body.departments}")`. In Spring XML you would use the `<simple>` tag instead of the `<method>` tag: `<simple>${body.departments}</simple>`.

The Splitter will usually operate on messages that are loaded into memory. But there are situations where 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 potentially be very large, so you have to split those files without loading the entire file into memory.

This can be done 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.

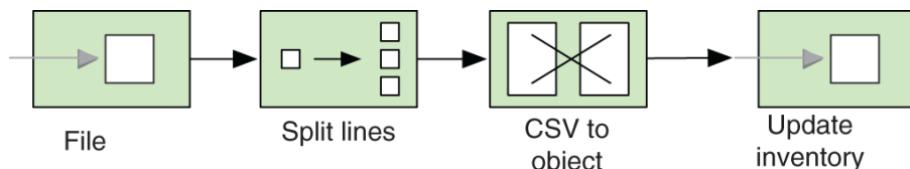


Figure 5.9 A route that picks up incoming files, splits them, and transforms them so they're ready for updating the inventory in the ERP system

We'll revisit this example again in TODO chapter 10, 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 follows.

Listing 5.5 Splitting big files 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() ①
            .bean(InventoryService.class, "csvToObject")
            .to("direct:update")
        .end()
        .log("Done processing big file: ${header.CamelFileName}");
    from("direct:update")
        .bean(InventoryService.class, "updateInventory");
}
```

- ① Splits file using streaming mode
- ② Denotes where the splitting route ends

As you can see in listing 5.5, all you have to do is enable streaming mode using `.streaming()` ①. 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()` ② to indicate the end of the splitting route. The `end()` in the Java DSL is the equivalent of the `</split>` tag when using Spring XML.

In Spring XML, you enable streaming using the `streaming` attribute on the `<split>` tag, as follows.

Listing 5.6 Splitting big files using streaming mode in Spring 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 using a tokenizer. The tokenizer is a powerful feature that works well with streaming. The tokenizer leverages

`java.util.Scanner`, which supports streaming. The `Scanner` is capable of iterating, which means that it only 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. So, in this example, the `Scanner` will only 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 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 are 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.

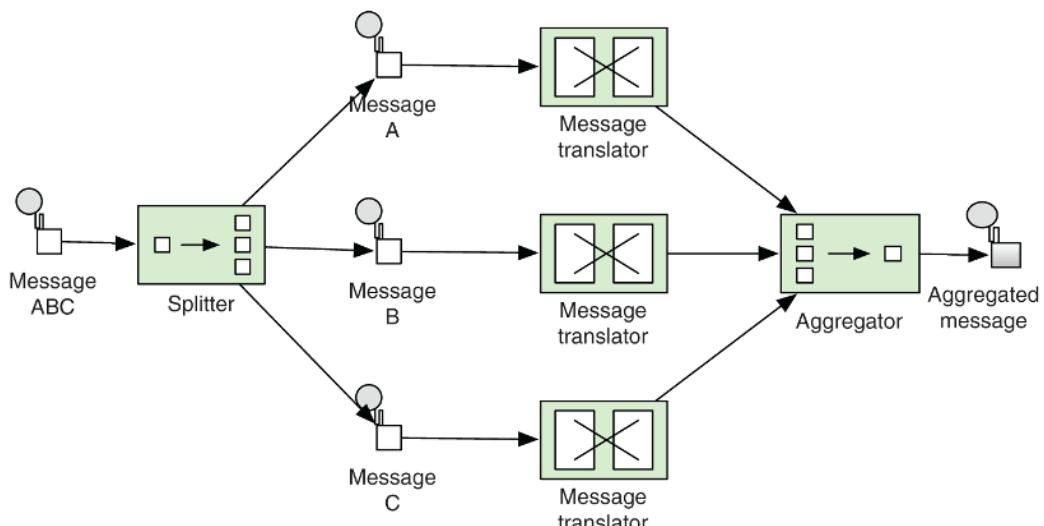


Figure 5.10 The Splitter has a built-in aggregator that can recombine split messages into a combined outgoing message.

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. This can easily be done 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 Spring XML, you have to declare the `AggregationStrategy` as a Spring bean tag, as shown in bold:

```
<bean id="translate" class="camelinaction.WordTranslateBean"/>
<bean id="myAggregationStrategy"
      class="camelinaction.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 the `AggregationStrategy`.

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);
        String existing = oldExchange.getIn().getBody(String.class);
        oldExchange.getIn().setBody(existing + "+" + body);
        return oldExchange;
    }
}
```

As you can see from listing 5.7, 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 some 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 Camel's error handling couldn't handle.

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 the `AggregationStrategy` and decide whether or not 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 Spring XML, you use the `stopOnException` attribute on the `<split>` tag, as follows:

```
<split strategyRef="myAggregationStrategy" stopOnException="true">
```

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=SplitterStopOnExceptionABCTest
mvn test -Dtest=SpringSplitterStopOnExceptionABCTest
```

The second option is to handle exceptions from the split messages in the AggregationStrategy.

HANDLING EXCEPTIONS USING AGGREGATIONSTRATEGY

The `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; ①
        }
        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;
    }
}
```

① Ignores the exception

When handling exceptions in the `AggregationStrategy`, you can detect whether an exception occurred or not by checking the `getException` method from the `newExchange` parameter. The preceding example ignores the exception by returning the `oldExchange` ①.

If you want to propagate back the exception, you need to keep it stored on the aggregated exception, which can be done as follows.

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());
        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;
}
```

1 Propagates exception

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 simply return the `newExchange` (which has the exception). Otherwise you must transfer the exception to the `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 have handled the exception and will ignore it.

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=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 TODO chapter 10 when we look at concurrency. In the next two sections, we'll look at EIPs that support dynamic routing, starting with the Routing Slip pattern.

5.4 The Routing Slip EIP

There are times when 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 using the Pipes and Filters EIP together with the Filter EIP. But as often happens with EIPs, there's a better way, known as 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.

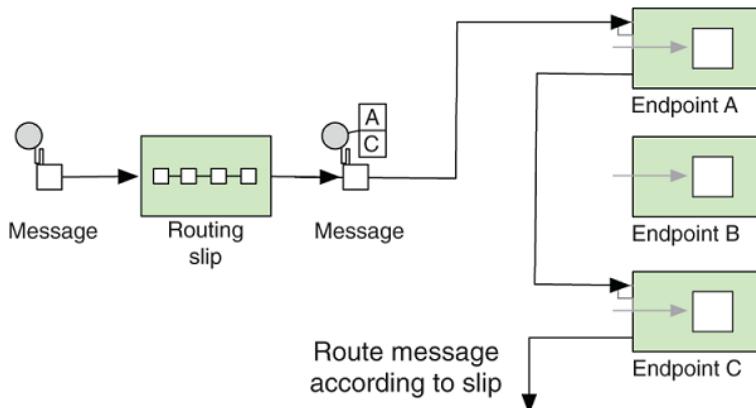


Figure 5.11 The incoming message has a slip attached that specifies the sequence of the processing steps. The Routing Slip EIP reads the slip and routes the message to the next endpoint in the list.

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 Spring XML:

```
<route>
    <from uri="direct:start"/>
    <routingSlip>
        <header>mySlip</header>
    </routingSlip>
</route>
```

This example assumes 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 Spring 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, we look at how to compute the header 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 leverage 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 Spring 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 is 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 we could use a method call expression like we 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 Spring XML is as follows:

```
<route>
    <from uri="direct:start"/>
    <routingSlip>
        <method beanType="camelinaction.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. By doing this, Camel will double up using the Routing Slip EIP, which is not the intention. Instead, do as shown in the example below.

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 Spring DSL:

```
<bean id="myBean" class="camelinaaction.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 a endpoint URI. Any Camel client or route could easily send a message to the bean and have it continued being routed as a routing slip.

For example, using a `ProducerTemplate` you could send a message to the bean:

```
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 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. So 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 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.

Listing 5.10 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
        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
        }
    }
}
```

- ① Previous endpoint URI
- ② Ends router

The idea with the Dynamic Router is to let Camel keep invoking the `route` method until it indicates the end. 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.

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 Spring XML is just as easy as shown:

```
<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 source code for the book contains this example in the `chapter5/dynamicrouter` directory. You can try it using the following Maven goals:

```
mvn test -Dtest=DynamicRouterTest
mvn test -Dtest=SpringDynamicRouterTest
```

There is also a Dynamic Router annotation you can use.

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, just 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 should not 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 Spring 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` its important to not use `dynamicRouter` in the DSL at the same time. Instead do as shown above.

The source code for the book contains this example in the chapter5/dynamicrouter directory. You can try it 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 was not distilled in the EIP book, but it will likely be added if there is a second edition of the book.

In this section, we’ll introduce the Load Balancer EIP by walking through an example. Then, in section 5.6.2, we’ll look at the various types of load balancers Camel offers out of the box. We’ll focus on the failover type in section 5.6.3 and finally show how you can 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. The `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.

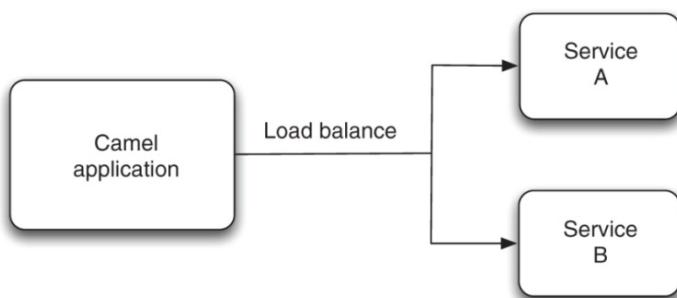


Figure 5.12 A Camel application load balances across two services.

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 you can 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 Spring 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 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 source code for the book 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
```

In the next section, we'll review the various load-balancing strategies you can use with the Load Balancer EIP.

5.6.2 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 different 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'll cover this in section 5.6.3. |
| Custom | Uses your own custom strategy. This is covered in section 5.6.4. |
| Circuit Breaker | A stateful pattern that monitors calls to processors for a specified exception. If the exception occurs enough the circuit breaker will trip and reject any new calls until a timeout is reached. |

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 Spring XML:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <random/>
        <to uri="seda:a"/>
        <to uri="seda:b"/>
    </loadBalance>
</route>
```

The sticky strategy requires 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 chose the same processor over and over again.

In the Java DSL, you would provide the expression using a header expression as shown here:

```
from("direct:start")
    .loadBalance().sticky(header("type"))
        .to("seda:a").to("seda:b")
    .end();
```

In Spring 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 source code for the book contains examples of using the strategies listed in table 4.9 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'll 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 Spring XML is as follows:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <failover/>
        <to uri="direct:a"/>
        <to uri="direct:b"/>
    </loadBalance>
</route>
```

The source code for the book contains this example in the chapter5/loadbalancer directory. You can try it 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 failover 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 only want to failover 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 is configured in Spring 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 Spring 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

| Configuration option | Default | Description |
|-------------------------|---------|---|
| maximumFailoverAttempts | -1 | Specifies how many failover attempts to try before exhausting (giving up): <ul style="list-style-type: none"> • Use -1 to attempt forever (never give up). • Use 0 to never failover (give up immediately). • Use a positive value to specify a number of attempts. For example, a value of 3 will try up to 3 failover attempts before giving up. |
| inheritErrorHandler | true | Specifies whether or not Camel error handling is being used. When enabled, the load balancer will let the error handler be involved. If disabled, the load balancer will failover immediately if an exception is thrown. |
| roundRobin | false | Specifies whether or not the load balancer operates in round robin mode. |

To better understand the options in table 5.10 and how the round robin mode works, we'll 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 failover 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 failover 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 Spring XML, you configure the options as attributes on the `failover` tag:

```
<route>
  <from uri="direct:start"/>
  <loadBalance>
    <failover roundRobin="true" maximumFailoverAttempts="1"/>
    <to uri="direct:a"/>
    <to uri="direct:b"/>
  </loadBalance>
</route>
```

The source code for the book contains this example in the `chapter5/loadbalancer` directory. You can try it using the following Maven goals:

```
mvn test -Dtest=FailoverLoadBalancerTest
mvn test -Dtest=SpringFailoverLoadBalancerTest
```

If you're curious about the `inheritErrorHandler` configuration option, take a look at the following examples in the source code for the book:

```
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.

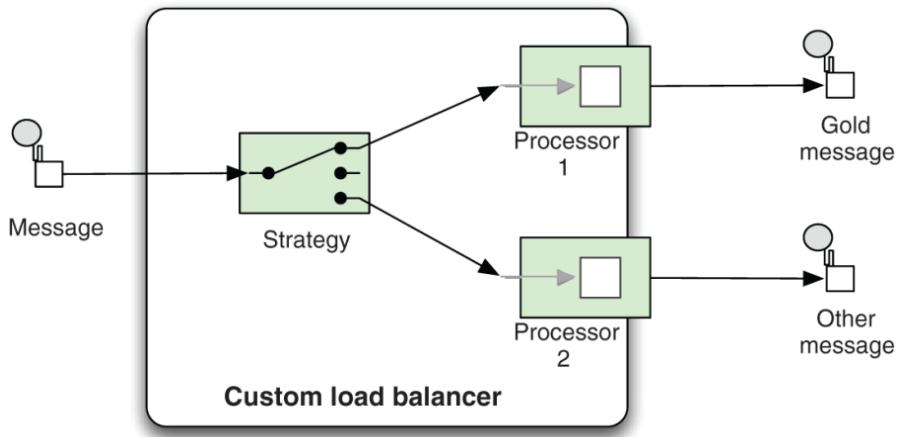


Figure 5.13 Using a custom load balancer to route gold messages to processor 1 and other messages to processor 2

When implementing a custom load balancer, you will often extend the `SimpleLoadBalancerSupport` class, which provides a good starting point. Listing 5.11 shows how you can implement a custom load balancer.

Listing 5.11 Custom load balancer

```
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.processor.loadbalancer.SimpleLoadBalancerSupport;
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);
        } else {
            return getProcessors().get(1);
        }
    }
}
```

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 will 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 Spring XML, you need to declare a Spring `bean` tag:

```
<bean id="myCustom" class="camelinaction.MyCustomLoadBalancer"/>
```

Which you then refer to from the `<loadBalance>` tag:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <custom ref="myCustom"/>
        <to uri="seda:a"/>
        <to uri="seda:b"/>
    </loadBalance>
</route>
```

The source code for the book contains this example in the chapter5/loadbalancer directory. You can try it using the following Maven goals:

```
mvn test -Dtest=CustomLoadBalancerTest
mvn test -Dtest=SpringCustomLoadBalancerTest
```

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 have had a common vocabulary, graphical notation, and concepts for designing applications to tackle today's integration challenges. You have 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 we covered in chapter 2 and those we presented in this chapter. Consider getting the EIP book to read more about the patterns—there's great advice given in the book. 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/enterprise-integration-patterns.html>.
- *Start simply.* When learning to use an EIP, you should create a simple test to try out 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 expected.

- *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 very sophisticated and have many features and options to tweak.

The Transactional Client EIP is useful for controlling transactions, and it's the topic of the next chapter.

6

Using components

This chapter covers

- An overview of Camel components
- Working with files and databases
- Messaging with JMS
- Networking with Netty
- Sending and receiving email
- In-memory messaging
- Automating tasks with the Quartz and Scheduler components

So far, we've only touched on 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 see how components are added to Camel. Then, although we can't describe every component in Camel—that would at least triple the length of this book—we'll look at the most commonly used components.

Table 6.1 lists the components we'll cover in this chapter and lists the URLs for their official documentation.

Table 6.1 Components discussed 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 off 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 really grown. As of version 2.15.0, Camel ships with more than 180 components, and there are dozens more available separately from other community sites. These components allow you to bridge to many different 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.

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 traveling down a route will need to take an off ramp to get to another route or external service. If the message is headed for another route, it will need 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();
    EndpointConfiguration createConfiguration(String uri) throws Exception;
```

```
ComponentConfiguration createComponentConfiguration();
}
```

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 will hold a reference to the `CamelContext`. The `CamelContext` provides access to Camel's common facilities, like the registry, class loader, and type converters. TODO ref custom components chapter for other methods 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.

There are two main ways in which components are added to a Camel runtime: by manually adding them to the `CamelContext` and through autodiscovery.

6.1.1 Manually adding components

You've seen the manual addition of a component already. In chapter 2 **TODO what chapter**, you had to add a configured JMS component to the `CamelContext` to utilize a `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.jmsComponentAutoAcknowledge` 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.

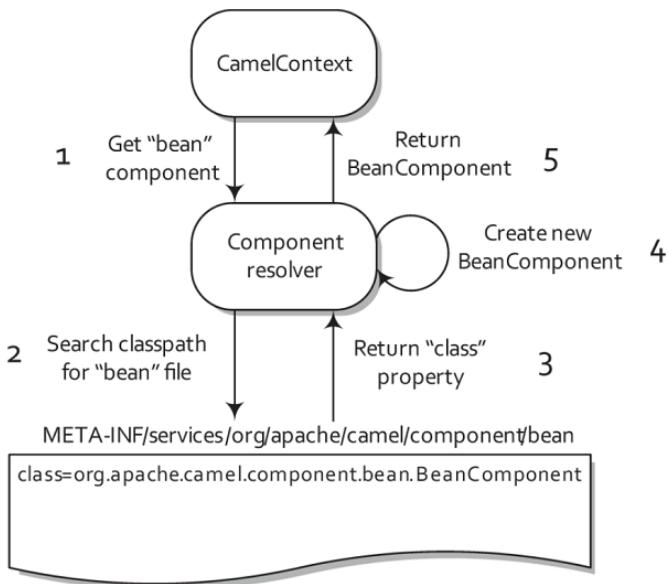


Figure 6.2 To autodiscover a component named “bean”, the component resolver searches for a file named “bean” in a specific directory on the classpath. This file specifies that the component class that will be created is BeanComponent.

Autodiscovery is the way the components that ship with Camel are registered. In order 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 what the name of a component is and what the fully qualified class name is.

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’ll discuss how to create your own Camel component in section 11.3 in chapter 11.TODO

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. We

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 |
| Binding | Binding allows you to wrap an endpoint with a contract. A binding can do things like marshal data using a data format or validate the message contents. | http://camel.apache.org/binding.html |
| Browse | Allows you to browse the list of exchanges that passed through a browse endpoint. This can be useful for testing, visualization tools, 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 |
| ControlBus | Allows you to control the lifecycle of routes and gather performance statistics by sending messages to a ControlBus endpoint. Based on the Control Bus EIP pattern. | http://camel.apache.org/controlbus-component.html |
| DataFormat | A convenience component that allows you to invoke a data format as a component. | http://camel.apache.org/dataformat-component.html |
| DataSet | Allows you to create large numbers of messages for soak or load testing. | http://camel.apache.org/dataset.html |
| Direct | Allows you to synchronously call another endpoint with little overhead. We'll look at this component in section 6.x. | http://camel.apache.org/direct.html |
| Direct-VM | Allows you to synchronously call another endpoint in the same JVM. We'll discuss this component in section 6.x. | http://camel.apache.org/direct-vm.html |
| File | Reads or writes to files. We'll discuss this component in section 6.x. | http://camel.apache.org/file2.html |
| Language | Executes a script against the incoming exchange using one of the languages supported by Camel. | http://camel.apache.org/language-component.html |

| | | |
|------------|--|---|
| Log | Logs messages to a number of different logging providers. | http://camel.apache.org/log.html |
| Mock | Tests that messages flow through a route as expected. You will see the Mock component in action in chapter 9. | http://camel.apache.org/mock.html |
| Properties | Allows you to use property placeholders in endpoint URIs. You'll see this in chapter 9. | http://camel.apache.org/properties.html |
| Ref | Looks up endpoints in the registry. | http://camel.apache.org/ref.html |
| Stub | Allows you to stub out real endpoint URIs for development or testing purposes. | http://camel.apache.org/stub.html |
| SEDA | Allows you to asynchronously call another endpoint in the same CamelContext. We'll look at this component in section 6.x. | http://camel.apache.org/seda.html |
| Stub | Allows you to stub out real endpoint URIs for development or testing purposes. | http://camel.apache.org/stub.html |
| Test | Tests that messages flowing through a route match expected message pulled from another endpoint. You will see Test component in action in chapter 9. | http://camel.apache.org/test.html |
| Timer | Sends out messages at regular intervals. You'll learn more about the Timer component and a more powerful scheduling endpoint based on Quartz in section 6.x of this chapter. | http://camel.apache.org/timer.html |
| Validator | Validate the message body using the JAXP Validation API. | http://camel.apache.org/validation.html |
| VM | Allows you to asynchronously call another endpoint in the same JVM. We'll discuss this component in section 6.x. | http://camel.apache.org/vm.html |
| XSLT | Transform a message using an XSLT template. | http://camel.apache.org/xslt.html |

Now let's look at each component in detail. We'll start by looking at 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, we 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, 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.

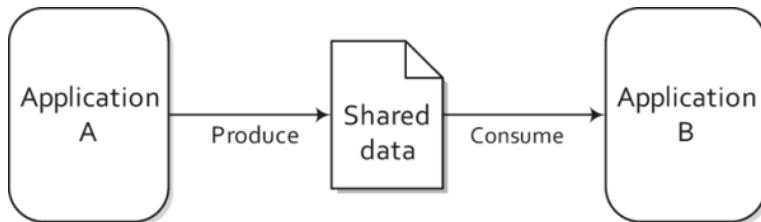


Figure 6.3 A file transfer between two applications is a common way to integrate with legacy systems.

Another reason why 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 IO APIs, platform-specific filesystem issues, concurrent access, and the like.

Camel has extensive support for interacting with filesystems. In this section, we'll look at how to use the File component to read files from and write them to the local filesystem. We'll also cover some advanced options for file processing and discuss how 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. Some common options are shown in table 6.3; 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

| Option | Default value | Description |
|-----------|---------------|---|
| delay | 500 | Specifies the number of milliseconds between polls of the directory. |
| recursive | false | Specifies whether or not to recursively process files in all subdirectories of this directory. |
| noop | false | Specifies file-moving behavior. By default, Camel will move 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. |
| fileName | null | 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. |

| | | |
|-----------|----------|---|
| fileExist | Override | Specifies what a file producer will do if the same filename already exists. Valid options are Override, Append, Fail, Ignore, Move, and TryRename. Override will cause the file to be replaced. Append adds content to the file. Fail causes an exception to be thrown. If Ignore is set, an exception won't be thrown and the file won't be written. Move will cause the existing file to be moved. Move also requires the moveExisting option to be set to an expression used to compute the filename to move to. TryRename will cause the a file rename to be attempted. TryRename only applies to temporary files specified by the tempFileName option. |
| delete | false | Specifies whether Camel will delete the file after processing. By default, Camel will not delete the file. |
| move | .camel | Specifies the directory to which Camel moves files after it's done processing them. |
| include | null | Specifies a regular expression. Camel will process only those files that match this expression. |
| exclude | null | Specifies a regular expression. Camel will exclude 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 pretty straightforward. Here's a simple example:

```
public void configure() {
    from("file:data/inbox?noop=true").to("stream:out");
}
```

This route will read files from the data/inbox directory and print 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 removed the `noop` flag and changed the route to the following:

```
public void configure() {
    from("file:data/inbox").to("stream:out");
}
```

This would use 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 would not 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 6.2.

By default, Camel will also lock any files that are being processed. The locks are released after routing is complete.

Both of the two preceding routes will consume any file not beginning with a period, so they will ignore files like `.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 will instruct 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 some text into the console, and press Enter, like this:

```
INFO Apache Camel 2.15.0 (CamelContext: camel-1) started in 0.276 seconds
Enter something:Hello
```

The example will keep running until you press Ctrl-C. The text (in this case, "Hello") will be read in by the Stream component and added as the body of a new message. This message's body (the text you entered) will then be 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-bender-35203-1427053605746-0-1
```

Because you did not 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 March 22, 2015, the file in the `data/outbox` directory will be named something like this:

```
20150322-02:00:53.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, there are a plethora of configuration possibilities listed in the online documentation.

We've started slowly with the File component, to get you comfortable with using components in Camel. Next we'll look at the FTP component, which builds on the File component but introduces messaging across a network. After that, we'll be getting 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
- SFTP for secure transfer
- FTPS (FTP Secure) 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 | null | 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 | null | 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 will transfer 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 an additional dependency to your project. If you use Maven, you just have to add the following dependency to your POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>2.15.0</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 for yourself, change to the `chapter6/ftp` directory and run this command:

```
mvn camel:run
```

After Camel has started, you'll need to enter something into the console:

```
INFO Apache Camel 2.15.0 (CamelContext: camel-1) started in 0.267 seconds
Enter something:Hello
```

The example will keep running until you press Ctrl-C.

You can now check to see if 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-bender-57493-1427055186933-0-1
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, like 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 we looked at JMS, it's also easy to use from Camel. In this section, we'll expand on what we covered 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 it 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 the `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 using TCP on port 61616 of the local host.

If you wanted to connect to a broker over some other protocol, ActiveMQ supports connections over VM, SSL, UDP, multicast, MQTT, AMQP, and so on. Throughout this section, we'll be demonstrating JMS concepts using ActiveMQ as the JMS provider, but any provider could have been used here.

THE ACTIVEMQ COMPONENT

By default, a JMS `ConnectionFactory` doesn't pool connections to the broker, so it will spin 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 configures connection pooling automatically for improved performance. The ActiveMQ component is used as follows:

```
<bean id="activemq"
[CA]class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
```

When using this component, you'll also need to depend on the `activemq-camel` module from ActiveMQ:

```
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-camel</artifactId>
    <version>5.11.1</version>
</dependency>
```

This module contains the ActiveMQ component and type converters for ActiveMQ data types.

Camel's JMS component has a daunting list of configuration options—over 80 to date. Many of these will only be seen in very specific JMS usage scenarios. The common ones are listed in table 6.5.

To use the JMS component in your project, you'll 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>2.15.0</version>
</dependency>
```

Table 7.5 Common URI options used to configure the JMS component

| Option | Default value | Description |
|-------------------------------|---------------|--|
| autoStartup | true | Controls whether consumers start listening right after creation. If set to <code>false</code> , you'll need to invoke the <code>start()</code> method on the consumer manually at a later time. |
| clientId | null | Sets the JMS client ID, which must be unique among all connections to the JMS broker. The client ID set in the <code>ConnectionFactory</code> overrides this one if set. |
| 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>JMSReplyToheader</code> in any messages or not. Set this if you don't want Camel to send a reply back to the destination specified in the <code>JMSReplyToheader</code> . |
| durableSubscriptionName | null | Specifies the name of the durable topic subscription. The <code>clientId</code> must also be set. |
| maxConcurrentConsumers | 1 | Sets the maximum number of consumer threads to use. If this value is higher than <code>concurrentConsumers</code> , new consumers are started dynamically as load demands. If load drops down, these extra consumers will be freed and the number of consumers will be equal to <code>concurrentConsumers</code> again. Increasing this value isn't advisable when using topics. |
| replyTo | null | Sets the destination that the reply is sent to. This overrides the <code>JMSReplyToheader</code> in the message. By setting this, Camel will use a fixed reply queue. By default, Camel will use a temporary reply queue. |
| replyToConcurrentConsumers | 1 | Sets the number of threads to use for request-reply style messaging. This value is separate from <code>concurrentConsumers</code> . |
| replyToMaxConcurrentConsumers | 1 | Sets the maximum number of threads to use for request-reply style messaging. This value is separate from <code>maxConcurrentConsumers</code> . |
| requestTimeout | 20000 | Specifies the time in milliseconds before Camel will timeout |

| | | |
|------------|-------|--|
| | | sending a message. You can override the endpoint value by setting the <code>CamelJmsRequestTimeout</code> header on a message. |
| selector | null | Sets the JMS message selector expression. Only messages passing this predicate will be consumed. |
| timeToLive | null | When sending messages, sets the amount of time the message should live. After this time expires, the JMS provider may discard the message. |
| transacted | false | Enables transacted sending and receiving of messages in InOnly mode. |

THE SJMS COMPONENT

Now, the Camel JMS component is built on top of the Spring JMS library so many of the options actually 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. So for memory-sensitive deployments or where you just aren't using Spring at all, Camel provides the SJMS component. The "S" in SJMS stands for Simple, Standard and Spring-less. While it isn't Camel's standard JMS component yet, it already provides much of the core functionality of the JMS component so it could replace `camel-jms` in the future.

To use this component, you'll need to add the `camel-sjms` to your Maven `pom.xml`:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sjms</artifactId>
  <version>2.16.0</version>
</dependency>
```

No other dependencies are required.

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. It started out as a step-by-step process: they 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 OK from accounting. A multicast EIP was used to implement this scenario.

Figure 6.4 illustrates another possible solution, which is to use a JMS topic following a publish-subscribe model. In that model, listeners such as accounting and production could

subscribe to the topic, and new orders would be published to the topic. In this way, both accounting and production would receive a copy of the order message.

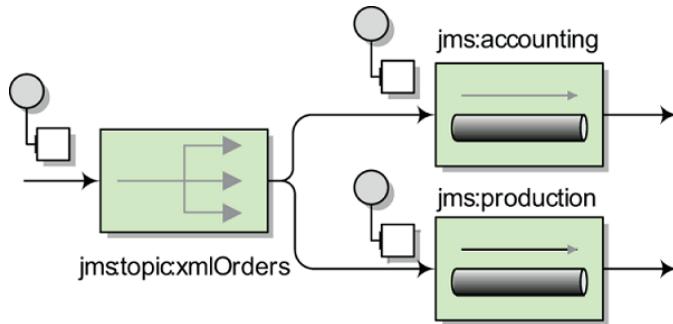


Figure 6.4 Orders are published to the xmlOrders topic, and the two subscribers (the accounting and production queues) get a copy of the order.

To implement this in Camel, you'd set up two consumers, which means there will be two routes 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 will receive a copy.

As you saw in chapter 2, an incoming order could originate from another route (or set of routes), like one that receives orders via a file, as shown in listing 6.1.

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
        .when(header("CamelFileName").regex("^.*(csv|csl)$"))
            .to("jms:topic:csvOrders");
from("jms:topic:xmlOrders").to("jms:accounting"); ②
from("jms:topic:xmlOrders").to("jms:production"); ②
```

- ① XML orders are routed to xmlOrders topic
- ② Both listening queues get copies

To run this example, go to the chapter6/jms directory in the book's source, and run this command:

```
mvn camel:run
```

This will output the following on the command line:

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

Why did you get this output? Well, you had a single order file named `message1.xml`, and it was published to the `xmlOrders` topic. Both the accounting and production queues were subscribed to the topic, so each received a copy. Testing routes consumed the messages on those queues and output the messages.

Messages can also be sent “by hand” to a JMS destination using a `ProducerTemplate`. A template class, in general, is a utility class that simplifies access to an API; in this case, the `Producer` interface. For example, to send an order to the topic using a `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. For more on the `ProducerTemplate`, see appendix C.

All of the JMS examples so far have been one-way only. Let’s look at how you can 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 there are times when 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 frontend 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 there are multiple replies awaiting. This flow of messages is illustrated in figure 6.5.

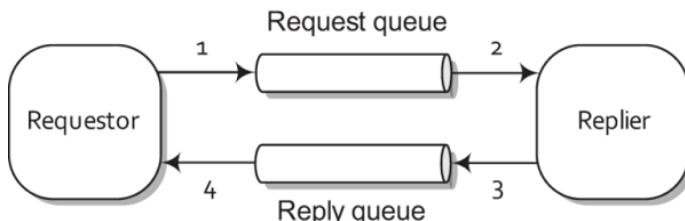


Figure 6.5 In request-reply messaging, a requestor sends a message to a request queue and then waits for a reply in the reply queue. The replier waits for a new message in the request queue, inspects the `JMSReplyTo` address, and then sends a reply back to that destination.

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 Rider Auto Parts' backend 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 just 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 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 also explicitly specify a reply queue. This can be done 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 will return the result of the `ValidatorBean`.

To try this out for yourself, go to the `chapter6/jms` directory in the book's source, and run this command:

```
mvn test -Dtest=RequestReplyJmsTest
```

The command will run a unit test demonstrating request-reply messaging as we've discussed in this section.

In the JMS examples we'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 what a message's body contains, JMS specifies different message types based on what the body type is. Figure 6.6 shows the five concrete JMS message implementations.

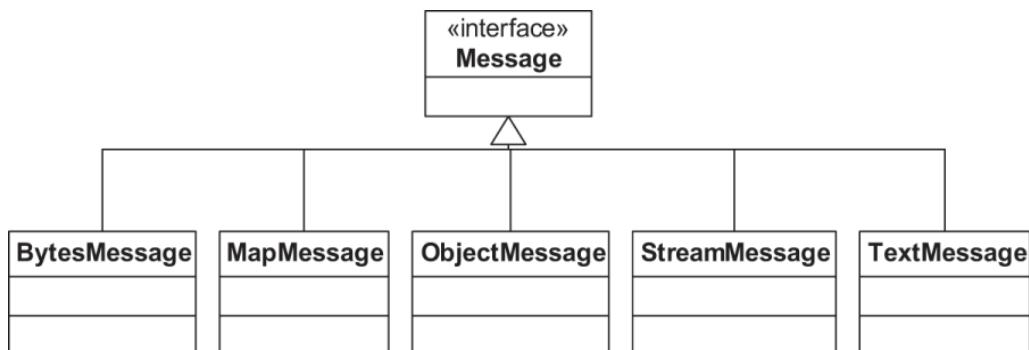


Figure 6.6 The `javax.jms.Message` interface has five implementations, each of which is built for a different body type.

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 will examine the body type and determine 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 |
|---|----------------------------|
| <code>String,org.w3c.dom.Node</code> | <code>TextMessage</code> |
| <code>byte[],java.io.File,java.io.Reader,java.io.InputStream,java.nio.ByteBuffer</code> | <code>BytesMessage</code> |
| <code>java.util.Map</code> | <code>MapMessage</code> |
| <code>java.io.Serializable</code> | <code>ObjectMessage</code> |

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 |
|----------------------------|----------------------------|
| <code>TextMessage</code> | <code>String</code> |
| <code>BytesMessage</code> | <code>byte[]</code> |
| <code>MapMessage</code> | <code>java.util.Map</code> |
| <code>ObjectMessage</code> | <code>Object</code> |
| <code>StreamMessage</code> | No mapping occurs |

Although this automatic message mapping allows you to utilize Camel's transformation and mediation abilities fully, you may sometimes need to keep the JMS message intact. An obvious reason would be to increase performance; not mapping every message means it will take 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 it, 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 a number of things. First, any headers that you set starting with "JMS" will be 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 will drop 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, like JMS and also FTP, run on top of other protocols. Let's look at how you can use Camel for these kinds of low-level communications.

TODO the entire CXF section was removed here. Need to incorporate the content into WS/REST chapter

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 like JMS and web services. All of 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 like 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 like TCP and UDP. In this section, we'll be using Netty to

demonstrate low-level network communication with Camel. For more information on using MINA with Camel, see the camel-mina2 component's documentation (<http://camel.apache.org/mina2.html>).

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>2.15.0</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 | null | 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 | null | 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, like <code>"#decoder1,#decoder2"</code> . |
| encoder | null | 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 | null | 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, like <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: LINE, or NULL. |
| sync | true | Sets the synchronous mode of communication. This means that clients |

| | | |
|------------------|-------------|--|
| | | will be able to get a response back from the server. |
| requestTimeout | 0 | Sets the time in milliseconds to wait for a response from a remote server. By default there is no timeout. |
| encoding | JVM default | Specifies the <code>java.nio.charset.Charset</code> used to encode the data. |
| transferExchange | false | Specifies whether only the message body is transferred. Enable this property to serialize the entire exchange for transmission. Only valid for TCP communications. |

In addition to the URI options, you also have to specify the transport type and port you want to use. In general, a Netty4 component URI will look like this,

```
netty4:transport://hostname:port[?options]
```

where `transport` is one of `tcp`, or `udp`.

Let's now see how you can 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 currently have floor personnel monitoring the machines manually. 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.

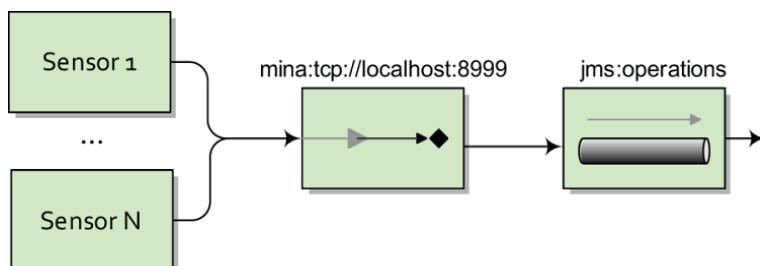


Figure 6.7 Sensors feed status messages over TCP to a server, which then forwards them to a JMS operations queue. TODO update diagram

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 using Netty, and it parses messages 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.

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.

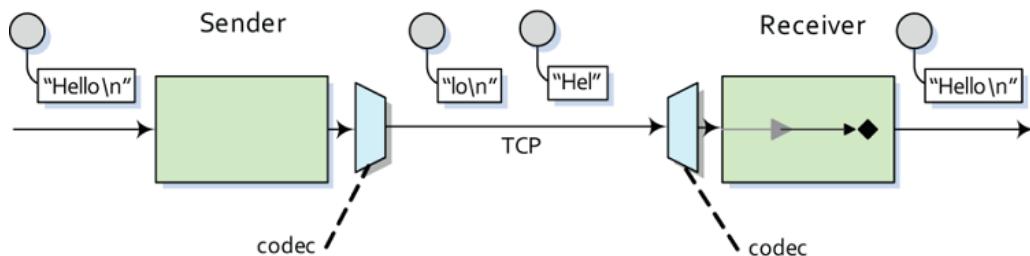


Figure 6.8 During TCP communications, a payload may be broken up into multiple packets. A `textline` codec can assemble the TCP packets into a full payload by appending text until it encounters a delimiter character.

This example is provided in the book's source in the `chapter6/netty` directory. Try it out using the following command:

```
mvn test -Dtest=NettyTcpTest
```

OBJECT SERIALIZATION CODEC

If you had not specified the `textline` URI option in the previous example, the Netty4 component would have defaulted to using the object serialization codec. This codec will take any `Serializable` Java object and send its bytes over TCP. This is a pretty handy codec if you aren't sure what payload format to use. If you're using this codec, you'll also need to ensure that the classes are on the classpath of both the sender and the receiver.

There are times when your payload will have a custom format that neither `textline` or object serialization accommodates. In this case, you'll 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. We can do this with a custom Netty codec.

In Netty, a codec consists of two parts:

- `ChannelOutboundHandler`—The `ChannelOutboundHandler` has the job of taking an input payload and putting bytes onto the TCP channel. In this example, the sensor will be transmitting the message over TCP, so you don't have to worry about this too much, except for testing that the server works.
- `ChannelInboundHandler`—The `ChannelInboundHandler` 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.

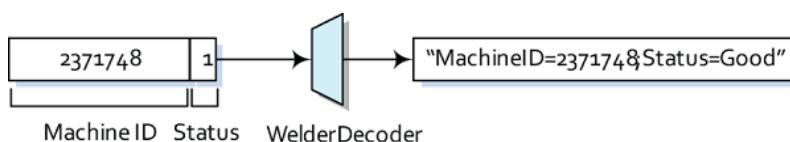


Figure 6.9 The custom welder sensor decoder is used to interpret an 8-byte binary payload and construct a plain text message body. The first 7 bytes are the machine ID and the last byte represents a status. In this case, a value of `1` means "Good".

Your route looks similar to the previous example:

```
from("netty4:tcp://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:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

```
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. If you 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 listing 6.3.

Listing 6.3 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(";") ❶
            .append("Status=");
            if (bytes[PAYLOAD_SIZE - 1] == '1') { ❷
                sb.append("Good");
            } else {
                sb.append("Failure");
            }
            out.add(sb.toString());
        } else {
            out.add(null);
        }
    }
}
```

- ❶ Gets first 7 bytes as machine ID
- ❷ Gets last byte as status

The decoder shown in listing 6.3 may look a bit complex, but it's only doing two main things: extracting the first 7 bytes and using that as the machine ID string ❶, and checking the last byte for a status of 1, which means "Good" ❷.

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 out low-level network communications, it's time to interact with one of the most common applications in the enterprise. That's the database.

6.5 Working with databases (JDBC and JPA components)

In pretty much every enterprise-level application, you'll 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 a number of 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 utilizing simple queries.
- *JPA component*—Persists Java objects to a relational database using the Java Persistence Architecture.
- *Hibernate component*—Persists Java objects using the Hibernate framework. This component isn't distributed with Apache Camel due to licensing incompatibilities. You can find it at the camel-extra project (<https://code.google.com/a/apache-extras.org/p/camel-extra/>). TODO ASF extras moving to github when google code shuts down?
- *MyBatis component*—Allows you to map Java objects to relational databases.

In this section, we'll be covering both the JDBC and JPA components. These are the most-used database-related components in Camel. 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 actual 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>2.15.0</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 <code>readSize</code> to be unbounded. |
| statement.propertyName | null | Sets the property with name <code>propertyName</code> on the underlying <code>java.sql.Statement</code> . |
| useHeadersAsParameters | false | Switch to using a <code>java.sql.PreparedStatement</code> with parameters that are replaced at runtime by message headers. This is a faster and safer alternative to executing a raw <code>java.sql.Statement</code> . PreparedStatements are precompiled so are faster and also are not susceptible to SQL-injection type attacks. |
| useJDBC4ColumnNameAndLabelSemantics | 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 actual 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. Of course, you can still retrieve data 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. So any incoming orders need to be put into the corporate database. Using Camel, a possible solution is illustrated in figure 6.9.

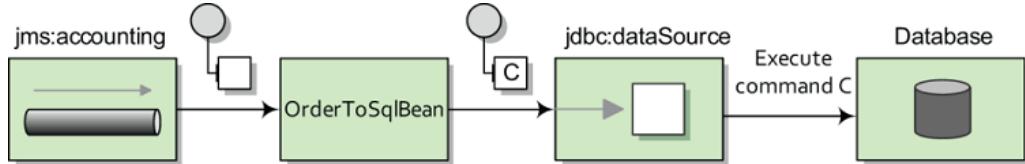


Figure 6.10 A message from the JMS accounting queue is transformed into an SQL command message by the OrderToSqlBean bean. The JDBC component then executes this command against its configured data source.

The main takeaway from figure 6.9 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 6.9 is simple on the surface:

```
from("jms:accounting")
  .to("bean:orderToSql")
  .to("jdbc:dataSource?useHeadersAsParameters=true");
```

There are several things that require explanation here. First, the JDBC endpoint is configured to load the `javax.sql.DataSource` with the name `dataSource` in the registry. The bean endpoint here is using the bean with the name `orderToSql` to convert the incoming message to an SQL statement and also to populate headers with information we'll be using in the SQL statement.

The `orderToSql` bean is shown in listing 6.4.

Listing 6.4 A bean that converts an incoming order to an 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 like:

```
partName = 'motor'
quantity = 1
customer = 'honda'
```

We 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. So 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 is not the default behavior of the JDBC component, we 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. So 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 (an SQL `select` command), Camel would return the rows as an `ArrayList<HashMap<String, Object>>`. Each entry in the `ArrayList` is a `HashMap` 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 for yourself, 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. There are times, though, that 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

There is a new requirement at Rider Auto Parts: 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 6.4, but there is a better solution for persisting objects.

The Java Persistence Architecture (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 pretty 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>2.15.0</version>
</dependency>
```

You'll also need to add JARs for the ORM product and database you're using. The examples in this section will 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>
  <version>2.3.0</version>
</dependency>
```

The JPA component has a number of URI options, many of which can only be applied to either a consumer or producer endpoint. The possible URI options are shown in table 6.9.

Table 6.9 Common URI options used to configure the JPA component

| Option | Consumer/ producer mode | Default value | Description |
|--------------------|----------------------------|---------------|--|
| persistenceUnit | Both | camel | Specifies the JPA persistence unit name used. |
| transactionManager | Both | null | Sets the transaction manager to be used. If transactions are enabled and this property isn't specified, Camel will use a <code>JpaTransactionManager</code> . TODO For more on transactions, see chapter 9. |
| maximumResults | Consumer | -1 | Specifies the maximum number of objects to be returned from a query. The default of -1 means an unlimited number of results. |

| | | | |
|-----------------------|----------|------|--|
| maxMessagesPerPoll | Consumer | 0 | Sets the maximum number of objects to be returned during a single poll. The default of 0 means an unlimited number of results. |
| consumeLockEntity | Consumer | true | Specifies whether the entities in the database will lock while they're being consumed by Camel. By default, they will lock. |
| consumeDelete | Consumer | true | Specifies whether the entity should be deleted in the database after it's consumed. |
| consumer.delay | Consumer | 500 | Sets the delay in milliseconds between each poll. |
| consumer.initialDelay | Consumer | 1000 | Sets the initial delay in milliseconds before the first poll. |
| 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 | | Causes objects that are sent to a JPA producer to be persisted to the underlying database immediately. 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. This means that your new POJO order class needs to have this annotation if you wish to persist it with JPA. The new order POJO is shown in listing 6.4.

Listing 6.4 An annotated POJO representing an incoming order

```
@Entity
public class PurchaseOrder implements Serializable {
    private String name;
    private double amount;
    private String customer;
    public PurchaseOrder() {
    }
```

1

```

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;
}
}
}

```

① Required annotation for objects to be persisted

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'll be using 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 listing 6.5.

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-3.0.xsd">

```

```

<bean id="jpa"
  class="org.apache.camel.component.jpa.JpaComponent">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="camel" />
  <property name="jpaVendorAdapter" ref="jpaAdapter" />
</bean>
<bean id="jpaAdapter"
  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"
  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>

```

- ① Hooks JPA component up to entity manager
- ② Creates entity manager
- ③ Uses OpenJPA and Apache Derby database
- ④ Allows JPA component to participate in transactions

The Spring beans file shown in listing 6.5 does a number of things to set up JPA. First off, 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 is one more thing left to configure before JPA is up and running. When the entity manager was created in listing 6.5 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. Listing 6.6 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>camelinaction.PurchaseOrder</class>
    <properties>
      <property name="openjpa.ConnectionDriverName"
        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="" />
    </properties>
  </persistence-unit>
</persistence>

```

```

<property name="openjpa.jdbc.SynchronizeMappings"
value="buildSchema" />
</properties>
</persistence-unit>
</persistence>
```

- ① Lists entity classes to be persisted
- ② Provides database connection information

There are two main things in listing 6.6 to be aware of. 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'll need to do so here ②.

Now that all of the setup is complete, your JPA route is complete. To try out 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 = (JpaEndpoint) context.getEndpoint("jpa:camelaction.PurchaseOrder");
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(0));

em.close();
```

First off, you create an `EntityManager` instance using the `EntityManagerFactory` from the `JpaEndpoint`. You then search for instances of your entity class in the database 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 is 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 different messaging abilities in this chapter, you might think there couldn't be more. Yet there is 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 is 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 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 Matt Welsh's SEDA page at <http://www.eecs.harvard.edu/~mdw/proj/seda/>.

Let's look first at the Direct components.

6.6.1 Synchronous messaging with the Direct and Direct-VM components

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
```

The only 3 options are listed in Table 6.10.

Table 6.10 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. |
| timeout | 30000 | The timeout in milliseconds for blocking the producer in the case where there were no active consumers. |
| failIfNoConsumers | true | Specifies whether to throw an exception when a producer tries to send to an endpoint with no active consumers. Only used when block is set to false. |

So 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 will invoke a web service defined by the `orderEndpoint` CXF endpoint bean. Let's also say that you send a message to this endpoint using a `ProducerTemplate`:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

```
String reply =
    template.requestBody("direct:startOrder", params, String.class);
```

The `ProducerTemplate` will create a `Producer` under the hood that sends to the `direct:startOrder` endpoint. In most other components, some processing happens in 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 is 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 there is little overhead to using the Direct component, 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), there are many benefits to using message queuing as a means of sending messages. You also saw how a routing application can be broken up into many logical pieces (routes) and connected by 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 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 is no message persistence to the disk, like 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 6.11.

Table 6.11 Common URI options used to configure the SEDA and VM components

| Option | Default value | Description |
|----------------------------------|------------------------|---|
| <code>size</code> | <code>Unbounded</code> | Sets the maximum number of messages the queue can hold. |
| <code>concurrentConsumers</code> | <code>1</code> | Sets the number of threads servicing incoming exchanges. Increase this number to process more exchanges concurrently. |

| | | |
|-----------------------|-----------------|---|
| waitForTaskToComplete | IfReplyExpected | Specifies whether or not the client should wait for an asynchronous task to complete. The default is to only wait if it's an InOutMEP. Other values include Always and Never. |
| timeout | 30000 | Sets the time in milliseconds to wait for an asynchronous send to complete. A value less than or equal to 0 will disable the timeout. |
| multipleConsumers | 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 together to form a routing application. For example, recall the example presented in 6.3.1 where 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 together. 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 6.10.

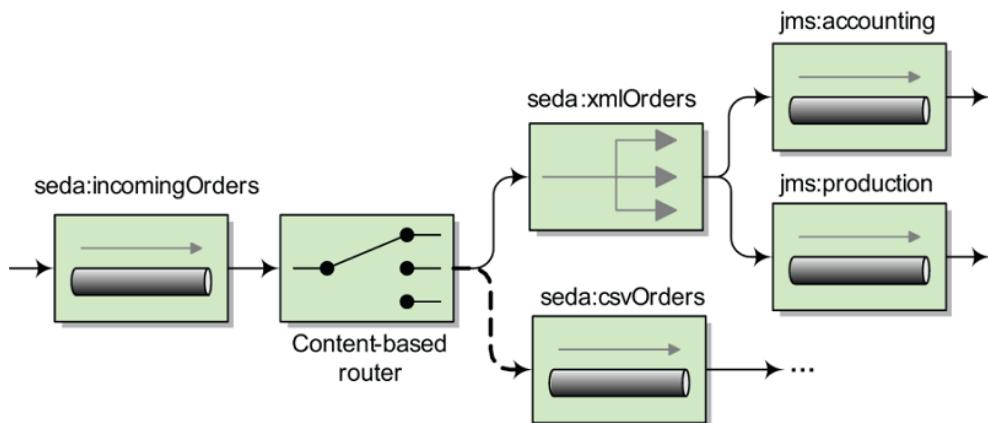


Figure 6.11 SEDA queues can be used as a low-overhead replacement for JMS when messaging is within a CamelContext. For messages being sent to other hosts, JMS can be used. In this case, all order routing is done via SEDA until the order needs to go to the accounting and production departments.

Any JMS messaging that you were doing within a 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 6.10. 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 listing 6.6.

Listing 6.6 A topic allows multiple receivers to get a copy of the message

```

from("file:src/data?noop=true")
    .to("seda:incomingOrders");
from("seda:incomingOrders")
    .choice()
        .when(header("CamelFileName").endsWith(".xml"))
            .to("seda:xmlOrders?multipleConsumers=true") ①
        .when(header("CamelFileName").regex("^.*(csv|csl)$"))
            .to("seda:csvOrders?multipleConsumers=true");
from("seda:xmlOrders?multipleConsumers=true") ②
    .to("jms:accounting");
from("seda:xmlOrders?multipleConsumers=true") ③
    .to("jms:production");

```

- ① Orders enter set of routes
- ② XML orders are routed to xmlOrders topic
- ③ Both listening queues get copies

This example behaves in the same way as the example in section 6.3.1, except that it uses SEDA endpoints instead of JMS. Another important detail in this listing is that SEDA endpoint URIs that are reused in consumer and producer scenarios need to be exactly the same as each other. It's not enough to specify the correct SEDA queue name; you need to use the queue name and all options.

To run this example, go to the chapter6/seda directory in the book's source, and run this command:

```
mvn test -Dtest=OrderRouterWithSedaTest
```

This will output the following on the command line:

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

Why did you get this output? Well, you had a single order file named message1.xml, and it was published to the xmlOrders topic. Both the accounting and production queues were subscribed to the topic, so each received a copy. The testing routes consumed 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'll 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, Quartz, 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.

In this section, we're first going to look at the Scheduler component and then move 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 a `ScheduledExecutorService` from the JRE to generate message exchanges at regular intervals. This component only supports consuming, because sending to a scheduler doesn't really make sense.

Some common URI options are listed in table 6.12.

Table 6.12 Common URI options used to configure the Scheduler component

| Option | Default value | Description |
|----------------------------|---------------|---|
| <code>delay</code> | 500 | Specifies the time in milliseconds between generated events. |
| <code>initialDelay</code> | 1000 | Specifies the time in milliseconds before the first event is generated. |
| <code>useFixedDelay</code> | true | If true, there will be a delay 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 is configuring the underlying `scheduledExecutorService` to have the execution interval of 2000 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, 2000 milliseconds can be written as `2s`, meaning 2 seconds. 90000 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 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>).

7

Microservices

This chapter covers

- Microservices overview and characteristics
- Developing microservices with Camel
- WildFly Swarm and Camel
- Spring Boot and Camel
- Vert.x and Camel
- Designing for failures
- Netflix Hystrix circuit breakers

We wanted to bring you the most up to date material in your hands. Because the IT industry is evolving constantly we decided to wait and write chapter 7 and 17 until last. This chapter covers using microservices with Camel and chapter 17 is a natural continuation where we go even further and take the microservices world into the cloud by using container technology such as Docker and Kubernetes. However this chapter stays down to earth and is mainly focused on running locally on a single computer or laptop.

We initially had a lengthy chapter introduction to talk about how the software is eating the world, how business are being disrupted, and that new emerging business are doing this in a much more agile and faster way, where microservices are instrumental. However we do feel there are much better authors whom can explain this, and we put two names in the hat which you can find in the start of section 7.1.

The first section will talk about eight microservices characteristics which you should have in mind when reading the remainder of this chapter.

Section 7.2 goes into action and show you how to build microservices with Camel using various containers, starting with a plain standalone Camel and all the way to using popular

containers such as WildFly Swarm and Spring Boot. We also made room for a interlude with building microservices using vert.x which we think is a very interesting project.

Section 7.3 is covering how microservices can call each other. We changed the flow of this chapter by making this section into a use case at Rider Auto Parts where you build a prototype consisting of six different microservices.

The first part of section 7.4 covers which strategies you can use for building fault tolerant microservices. The second part continues the Rider Auto Part use-case and walks you through how to improve those six microservices to become a full functional fault tolerant system. As part of the exercise you learn how to use the popular Hystrix circuit breaker from the Netflix OSS stack, and see first hand how elegantly Camel integrates with it.

The chapter ends on a high note with some eye candy with live graph which charts the state of the circuit breakers.

We start from the beginning with section 7.1 by setting the stage and 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 not an exception. Do not mistake microservices as only a technology discussion. It is equally part organizational structure, culture, and human forces.

What we have done is to pick a number of technical characteristics which we will be covering in this book. If you are interested in the non-technical side of microservices we recommend among others the following material:

- *Building Microservices* published by O'Reilly authored by Sam Newman
- *Microservices for Java Developers* published by O'Reilly authored by Christian Posta (free download: <https://developers.redhat.com/promotions/microservices-for-java-developers/>)

There are eight characteristics we want to cover. Keep in mind the selection is our modest choice we see as most relevant to this book at the time of writing.

7.1.1 Microservices Characteristics

Microservices is not a new technical invention such as an operating system, programming language would be. Microservices is a new term that describes a style of software systems that adheres to certain characteristics and design principles. Microservices is an architectural style where an application or software system is composed of individual standalone services communicating using light weight protocols in an event based manner.

In the following we will outline a number of characteristics and common practices used for implementing microservices and what role Camel plays.

SMALL IN SIZE

The very fundamental principle of a microservice is hinted by its name - that a microservice is small (micro) in size. The mantra of a microservice is said to be small, nimble, and do one thing only and do that well. Its debatable how to measure a size of a microservice, is that by the number of code lines, number of classes, etc. A good measure of the size of a microservice would be a single person should be able manage all of that in his/her head.

Camel really shines in this area. Camel applications are inherently small in size. For example a Camel application that receives events over HTTP or JMS, unmarshal, transform, persist, and sending a response back can be around 50 - 100 lines of code. That is small enough to fit into one persons head, whom can build end to end testing, do code refactoring. And if the microservice is not longer needed then the code can be thrown away without losing hundreds of hours of implementing and coding being wasted.

HAVING TRANSACTIONAL BOUNDARIES

An application composed of multiple microservices are widely deployed in distributed systems. The consistency model of distributed systems is known as eventual consistency (https://en.wikipedia.org/wiki/Eventual_consistency).

Eventual Consistency and distributed systems

A distributed system maintains copies of its data on multiple machines in order to provide high availability and scalability. Data changes from one machine must be propagated to the other replicas. Since such changes does not happen instant (network overhead) there is a small window during which some of the copies have the latest data and others do not. In other words the the copies are mutuality inconsistent, however the changes will eventually be replicated to all copies, and hence the term eventual consistency. In non distributed systems there is no need for propagated as the data is always consistent (single copy) and eventual consistent bear no meaning.

In an event based system it becomes important to know where the event (message) currently belongs and transactional boundaries helps with that. For example a Camel based microservice which listens on a JMS queue to process messages which are then routed to another destination for further processing by another microservice. Ensuring transactional behavior across heterogeneous distributed systems is hard, but luckily Camel has great transactional capabilities (covered in chapter 9). Camel has components that can participate in transactions, transacted routes, error handlers, idempotent consumer, and compensating actions, all of which helps developers create microservices with transactional behavior.

SELF MONITORING

Services should expose information that describes the state of the service and how the service performs. Running microservices in production should allow easy management and monitoring. A good practice is to provide health check's in your microservices, accessible from

known endpoints which the runtime platform uses for constantly monitoring. Such kind of information is provided out of the box by Camel which you can make available over JMX, HTTP, and microservice containers such as Spring Boot and WildFly Swarm.

TIP Chapter 16 will cover all about management and monitoring with Camel.

DESIGNED FOR FAILURE

A consequence of building distributed systems 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 this as gracefully as possible.

Since a service can come and go at any time its important to be able to detect the failures quickly and if possible automatic restore the service on the runtime platform - hence self monitoring.

Camel has a lot of EIP patterns to help design with failures. The Dead Letter Channel EIP can ensure messages are not 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 Circuit Breaker EIP can be used to protect unresponsive upstream services from receiving further calls in a period of time while the circuit breaker is open. We will cover designing for failures and using the Circuit Breaker EIP in section 7.4.

HIGHLY CONFIGURABLE

A sound practice is to never hardcode or commit your environment specific configuration in your source code, such as username and passwords. Instead keep the configuration outside your application in external configuration files or environment variables. A good phrase to remember this mantra is as follows *separate config from code*.

By keeping configuration outside your application, allow to deploy the same application to different environments by only changing 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 becomes the common practice, where a docker image (with your application) safely moves between environments (test, staging, pre-prod and production, etc.) by applying environment specific configuration upon the docker image.

Camel applications are highly configurable where 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 re-compile your source code, to setup credentials, encryption, threading model and whatnot.

SMART ENDPOINTS AND DUMB PIPES

Over the last couple of decades we have seen different integration products based on the principles of EAI, ESB, and 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 would rely on overly complex protocols such as WS-Choreography, BPEL, or BPMN.

Microservices favor RESTish and light-weight messaging protocols rather than complex protocols such as WebServices. The style used by microservices is *smart endpoints and dumb pipes*. Applications built from microservices aims 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 and event, apply some logic, and return a response.

Camel supports both worlds. You can find components supporting WebServices, RPC, BPMN, JMS, etc. For microservices you can find a lot of components supporting REST, as well a Rest DSL which makes defining RESTful services much easier (we cover this in chapter 10). There is plenty of components for light-weight messaging such as AMQP, Amazon SQS/SNS, Kafka, MQTT, Stomp, etc. And each new Camel release bring in more components.

TESTABLE

Lets step back and ask ourselves why are we doing microservices? There can be many reasons for doing so, but a key goal should be about being able to deliver services faster into production. By having many more microservices being deployed into production, then we need developers and operations teams to have confidence that what is being deployed are working as intended - and to help ensure this testing microservices should be faster and easier to do as well - and if possible as automated as possible.

Camel has extensive support for testing at different levels such as unit and integration tests. With Camel you can test your application in isolation by mocking external endpoints, simulating events, and verifying all is as expected. We have devoted an entire chapter 9 to cover all this.

INFRASTRUCTURE AUTOMATION

Infrastructure has evolved tremendously over the last couple of years. For example the three cloud provider giants Amazon AWS, Microsoft Azure, and Google GKE has push the bar higher making it easier for developers and operations to deploy and run their applications in their infrastructure. Companies whom want to take this experience on-premise can purchase cloud software from different vendors such as CoreOS, Docker Inc, Red Hat, and Pivotal.

A great use-case for using infrastructure automation is a CI/CD system. Teams within your organization should then leverage the infrastructure to have an automated CI/CD pipeline where code changes are build and testing automatically. This gives fast and continues feedback to the developers whom can catch and correct mistakes as early as possible.

Moreover with teams owning and operating its own microservices, we need a way for teams to repetitively deliver their services in a fast and reliable manner. Teams should also have access to insights and usages of their microservices as a constant feedback loop. The feedback and increasing changes to business requirements should foster teams to do more

releases more often. To make this a reality we need a CI/CD pipeline system. These pipelines may be composed of multiple sub-pipelines with gates and promotions steps, but ideally we should automate the build, test and deploy mechanism as much as possible.

We will touch point about these microservices characteristics in the remainder of the chapter, except for infrastructure automation, which will be part of chapter 17.

It's time to leave the background theory and 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? And the answer is all of them. Camel is just a library you include in the JVM runtime and it runs anywhere. In this section we will walk you through how to run Camel in some of the most popular microservice runtimes:

- Standalone - Running just Camel
- CDI - Running Camel with CDI container
- WildFly Swarm - Lets see how Camel runs with the light-weight JEE server
- Spring Boot - Running Camel with Spring Boot
- Vert.x - Running Camel with the reactive vert.x platform

As you can see there is seven runtimes in the bulleted list, so there is a lot to cover. We start with just Camel and then walk our way down the list and ending with some of the never and more eccentric microservice runtimes.

7.2.1 Standalone Camel as microservice

Throughout section 7.2 we will use a basic Camel example as the basis of a microservice. The service is a 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 using standalone Camel is done using the following three steps:

- Create a `CamelContext`
- Add the route with the service to the `CamelContext`
- Start the `CamelContext`

And voila you have Camel running standalone as a microservice as shown below:

```
public class HelloCamel {

    public static void main(String[] args) throws Exception {
```

```

CamelContext context = new DefaultCamelContext();
context.addRoutes(new HelloRoute());
context.start();

Thread.sleep(Integer.MAX_VALUE);
}
}

```

1
2
3

4

- ① Create the CamelContext
- ② Add the route with the service
- ③ Start Camel
- ④ Keep Camel (the JVM) running

You can try this example from the source code in the chapter7/standalone directly 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 ④ we had to use `Thread.sleep` to keep Camel running. Whenever there is a code smell there usually is a better way with Camel.

RUNNING CAMEL STANDALONE USING CAMEL MAIN CLASS

Camel provides a `org.apache.camel.main.Main` class which makes it easier to run Camel standalone as shown below:

```

public class HelloMain {

    public static void main(String[] args) throws Exception {
        Main main = new Main();
        main.addRouteBuilder(new HelloRoute());
        main.run();
    }
}

```

1
2
3

- ① Create the Main class
- ② Add the route with the service
- ③ Keep Camel (the JVM) running

As you can see running Camel using the `Main` class is easier. The `run` method ③ is a blocking method that keeps the JVM running. The `Main` class also ensures to trigger on JVM shutdown signals and perform a graceful shutdown of Camel so your Camel applications terminates graceful.

You can try this example from the source code by running the following Maven goal:

```
mvn compile exec:java -Pmain
```

To terminate the JVM you can press `ctrl + c`, and notice from the console log that Camel is stopping. This did not happen with the previous example, where we do not ensure to call the `stop` method on `CamelContext`.

The `Main` class has additional methods for configuration such as property placeholders and to register beans in the registry. You can find details about this 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. You can quickly get started but it has its limits. For example you would need to figure out how to package your application code together with the needed JARs from Camel and 3rd party dependencies, and how to run that as a Java application. You could try to build a fat jar but that has its problems if there are duplicate files that need to be merged together.

However if you are using Docker containers you can actually package your application together in a docker image with all the JARs and your application code separated, and still make it very easy to run your application. We will be covering Camel and Docker in chapter 17.

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 will be covering in sections to follow. But first lets talk about Camel and CDI before we get to WildFly Swarm.

7.2.2 CDI Camel as microservice

So what is CDI? CDI (Content Dependency Injection) is a Java specification for dependency injection including a set of key features such as:

- 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 container to customize behavior.

Camel provides support for CDI by the `camel-cdi` component which contains a set of CDI extensions that make it very easy to setup 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 show in listing 7.1

Listing 7.1 - Camel route using CDI

```
package camelinaction;

import javax.inject.Singleton;
import org.apache.camel.builder.RouteBuilder;

@Singleton
public class HelloRoute extends RouteBuilder {1

    public void configure() throws Exception {
        from("jetty:http://localhost:8080/hello")
            .transform().simple("Hello from Camel");
    }
}
```

① Define scope of bean as Singleton

As you can see from the source code in listing 7.1, the Camel route is just regular Java DSL code without any special CDI code. In fact the only code from CDI is the `@Singleton` annotation ①.

To make running Camel in CDI easier there is a `org.apache.camel.cdi.Main` class we can use as shown below:

```
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 very simple, lets push the bar a little and configure Camel using CDI.

CONFIGURING CDI APPLICATIONS

The code in listing 7.1 returns a reply message that is hardcoded. Lets improve this and externalize the configuration to a properties file. Camel's property placeholder mechanism is a component with the `id` properties. In CDI you use `@Produces` and `@Named` annotations to create and configure beans. We use this to create and configure an instance of `PropertiesComponent` as shown in listing 7.2.

Listing 7.2 - Configuring Camel property placeholder using CDI @Produces

```
package camelinaction;

import javax.enterprise.inject.Produces;
import javax.inject.Named;
import javax.inject.Singleton;

@Singleton
public class HelloConfiguration {

    @Produces
    @Named("properties")
    PropertiesComponent propertiesComponent() {
        PropertiesComponent component = new PropertiesComponent();
        component.setLocation("classpath:hello.properties");
        return component;
    }
}
```

1
2
3

- ① Declare this method is producing (creating) a bean
- ② ... with the name properties
- ③ ... and of type PropertiesComponent

In listing 7.2 we have created a class named `HelloConfiguration` which we use to configure our application using CDI. We think this is good practice to have one or more configuration classes separated from your business and Camel routing logic.

To setup Camel's property placeholder we use a java method that creates, configures and returns an instance of `PropertiesComponent` ③ . Notice how the code in this method is plain Java code without using CDI. Its only on the method signature we use CDI annotations to instruct CDI that this method is able to produce ② a bean of type `PropertiesComponent` with the id `properties` ③ .

This example is shipped with the source code of the book in the `chapter7/cdi` directory which you can try using the following Maven goal:

```
mvn compile exec:java
```

Lets raise the bar one more time and show you the most used feature of CDI, how you can use CDI dependency injection in your POJO beans using `@Inject`.

DEPENDENCY INJECTION USING @INJECT

We have refactored the example to use a POJO bean to construct the reply message of the hello service as shown below:

```
import javax.inject.Singleton;
import org.apache.camel.PropertyInject;
import org.apache.camel.util.InetAddressUtil;

@Singleton
public class HelloBean {
```

```

public String sayHello(@PropertyInject("reply") ① String msg)
    throws Exception {
    return msg + " from " + InetAddressUtil.getLocalHostName();
}
}

```

① Inject Camel property placeholder with key reply as parameter

The `HelloBean` class has a single method named `sayHello` which creates the reply message. The method takes one argument as input which has been annotated with Camel's `@PropertyInject` annotation. `@PropertyInject` is configured with the value `reply`, which corresponds to the property key. The property placeholder file should contain this key such as:

```
reply=Hello from Camel CDI with properties
```

The Camel route for the hello service needs to be changed to use the `HelloBean` which can be done by dependency injecting the bean into the `RouteBuilder` class as shown in listing 7.3.

Listing 7.3 - Injecting bean using CDI @Inject

```

import javax.inject.Inject;
import javax.inject.Singleton;
import org.apache.camel.builder.RouteBuilder;

@Singleton
public class HelloRoute extends RouteBuilder {

    @Inject
    private HelloBean hello; ①

    public void configure() throws Exception {
        from("jetty:http://localhost:8080/hello")
            .bean(hello, "sayHello"); ②
    }
}

```

① Dependency inject the HelloBean using @Inject

② Call the method sayHello on the injected HelloBean instance

Because we want to use the `HelloBean` in the Camel route, we 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 source code of the book in the chapter7/cdi directory. With the source code you see how to use `@Inject` in unit tests with CDI. We have devoted the entire chapter 9 on the topic of testing where we will cover testing CDI in much more details.

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 as shown below:

```
import javax.inject.Inject;
import javax.inject.Singleton;
import org.apache.camel.Endpoint;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.cdi.Uri;

@Singleton
public class HelloRoute extends RouteBuilder {

    @Inject
    private HelloBean hello;

    @Inject @Uri("jetty:http://localhost:8080/hello") ①
    private Endpoint jetty;

    public void configure() throws Exception {
        from(jetty) ②
            .bean(hello, "sayHello");
    }
}
```

- ① Dependency inject Camel endpoint with the given uri
- ② Use the injected Camel endpoint in the Camel route

In the `HelloRoute` class we use a field to define the Camel endpoint for the incoming endpoint

① . Notice how the field is annotated with both `@Inject` (CDI) and `@Uri` (`camel-cdi`). In the Camel route the endpoint is used in the `from` which accepts an endpoint as parameter type ②

You can also use `@Inject @Uri` to inject a `FluentProducerTemplate` which makes it easy to send a message to the given endpoint. We use this in unit testing this example as shown in listing 7.4.

Listing 7.4 Using @Inject @Uri to inject FluentProducerTemplate

```
import javax.inject.Inject;
import org.apache.camel.FluentProducerTemplate;
import org.apache.camel.cdi.Uri;
import org.apache.camel.test.cdi.CamelCdiRunner;
import org.junit.Test;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertTrue;

@RunWith(CamelCdiRunner.class) ①
public class HelloRouteTest {

    @Inject @Uri("jetty:http://localhost:8080/hello") ②
    private FluentProducerTemplate producer;
```

```

    @Test
    public void testHello() throws Exception {
        String out = producer.request(String.class);      ③
        assertTrue(out.startsWith("Hello from Camel CDI"));
    }
}

```

- ① Unit testing with camel-cdi
- ② Dependency inject FluentProducerTemplate with default uri
- ③ Sending (InOut) empty message to the endpoint and receive 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 a `FluentProducerTemplate` ② to send a test message to the Camel hello service ③ . Notice how 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
private FluentProducerTemplate producer;          ①

@Test
public void testHello() throws Exception {
    String out = producer.to("jetty:http://localhost:8080/hello")   ②
        .request(String.class);
    assertTrue(out.startsWith("Hello from Camel CDI"));
}

```

- ① Inject template without a default endpoint uri
- ② Specify the uri of the endpoint to send the message to

The `FluentProducerTemplate` is not 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. We have also provided a variant of the example which runs in Apache Karaf using CDI with OSGi which is located in the `chapter7/cdi-karaf` directory.

Before reaching the end of our mini covering of using Camel CDI let us show you one last feature of CDI which you can use with Camel - event listening.

LISTENING TO CAMEL EVENTS USING CDI

CDI supports an event notification mechanism which allows you to listen for certain events and react upon. This can be used to listen to Camel lifecycle events such as when Camel Context or routes starts or stops. The following code shows how to listen for when Camel has just been started:

```

import javax.enterprise.event.Observes;
import javax.inject.Singleton;

@Singleton

```

```
public class HelloConfiguration {

    void onContextStarted(@Observes CamelContextStartedEvent event) { ①
        System.out.println("*****");
        System.out.println("* Camel started " + event.getContext().getName());
        System.out.println("*****");
    }
}
```

① Listen for CamelContextStartedEvent to happen

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 where we covered using three of the most common key features from CDI and camel-cdi:

- Dependency Injection using `@Inject` and `@Produces`
- Injecting Camel endpoint using `@Uri`
- Bean lifecycle using scopes such as `@Singleton`
- Event listening using `@Observes`

Before moving on to the next Camel microservice runtime lets share some of our thoughts on using Camel CDI.

SUMMARY OF USING CAMEL CDI FOR MICROSERVICES

Developers wanting to build microservices using Java code can find value in using a dependency injection framework like CDI or Spring with Java Config. Camel users whom find the XML DSL attractive should not 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 Java development model using Java code and annotations to configure Java beans and specify their inter relationships.

When using CDI you need a CDI container at runtime such as JBoss Weld which we have been using in our examples. JBoss Weld is not primary intended as a standalone server but find its primary use-case as a component inside an existing application server such as WildFly, WildFly Swarm, or Apache TomEE. Attempting to building your application as a fat jar deployment and run with JBoss Weld is not as easy. Instead you should look at a more powerful application server such as WildFly Swarm or Spring Boot which is being covered in the following.

7.2.3 WildFly Swarm with Camel as microservice

WildFly Swarm is a JEE application server where 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 JEE applications.

Its easy to get started with WildFly Swarm in a new project as you basically setup your Maven pom.xml with:

- Import WildFly Swarm BOM (bill of materials) dependency
- Declare the WildFly Swarm dependencies you need (such as Camel, CDI, etc.)
- Add the WildFly Swarm Maven plugin which generates the *fat jar*.

You can very easily get started with a new project using the generator webpage as shown in figure 7.1.

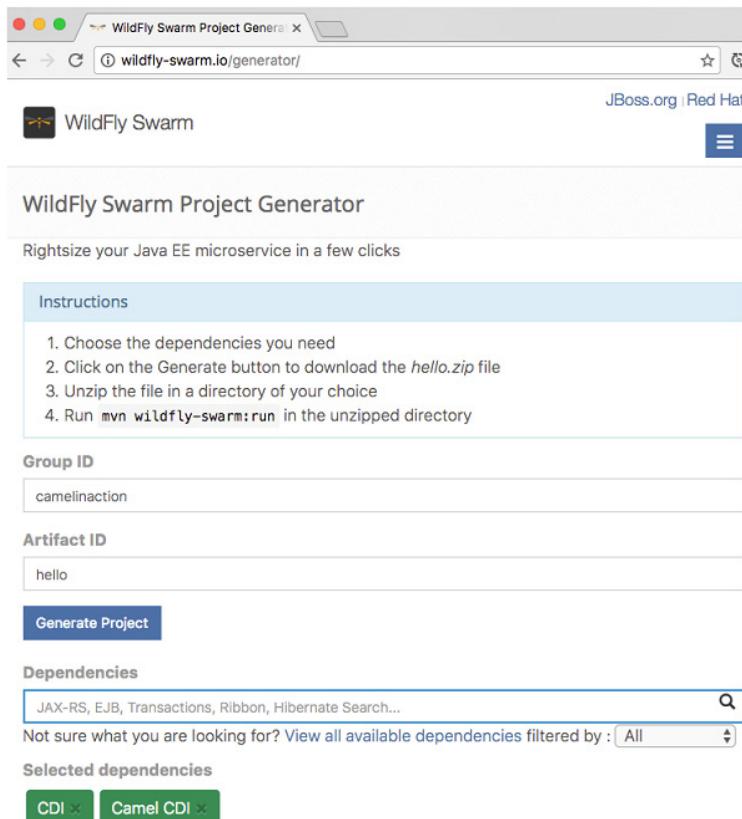


Figure 7.1 - Creating a new WildFly Swarm project from the generator webpage. Here we have chosen Camel CDI and CDI as the dependencies we need. Clicking the View all dependencies link will show all the dependencies you can chose among. Clicking Generate Project downloads a .zip file with the generated source code.

In the previous section we built a Camel microservice using CDI. Now we want to take that application and run on WildFly Swarm, and therefore we chose Camel CDI and CDI as dependencies. We then need to do one other change as we were using Jetty as the HTTP server, but WildFly Swarm comes out of the box with its own HTTP server named Undertow. This require us to change the source code as shown in listing 7.5.

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") ①
    private Endpoint undertow;

    @Override
    public void configure() throws Exception {
        from(undertow)
            .bean(hello, "sayHello");
    }
}
```

① Use Undertow as HTTP server as it comes out of the box with WildFly Swarm

And we also need to add camel-undertow as a dependency to our Maven pom.xml file as shown below:

```
<dependency>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>camel-undertow</artifactId>
</dependency>
```

Notice that the groupId of the dependency is not org.apache.camel but is org.wildfly.swarm. The reason is that WildFly Swarm provides a number of supported and curated dependencies. These dependencies are called fragments in WildFly Swarm. In this case there is a fragment for making Camel and Undertow work together, and hence we 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:

```
java -jar target/hello-swarm.jar
```

This was a brief coverage of using Camel with WildFly Swarm. However we covered most part 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 fragment over Camel component.

WILDFLY SWARM FRAGMENTS VERSUS CAMEL COMPONENTS

When using the JEE functionality from WildFly Swarm with Camel you should use the provided WildFly curated components, which is called fragment. In the example we used 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 is no specialized fragment for a Camel component you should use the regular component. For example there is 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 webpage, by clicking the View all available dependencies (as was shown in figure 7.1).

CDI is not the only option when using Camel on WildFly Swarm. You can also do 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 and therefore you can use Spring XML files to define Camel routes and beans. The route in listing 7.6 could be done in Spring XML as shown:

```
<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>
```

The location of the 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 source code of the book includes this example in the `chapter7/wildfly-camel-spring` directory which you can try 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 would need to add the monitor fraction to the Maven `pom.xml` file as shown:

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>monitor</artifactId>
</dependency>
```

You can try this by running the `chapter7/wildfly-swarm` example by

```
mvn wildfly-swarm:run
```

And 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. However at this time of writing there are no default health checks installed and the endpoint returns a HTTP Status 204:

```
$ curl -i http://localhost:8080/health
HTTP/1.1 204 No health endpoints configured!
```

However it is expected that the Camel fraction will include a Camel specific health check in a future release.

Lets end our coverage of WildFly Swarm by sharing our thoughts of the good and bad with using Camel with WildFly Swarm.

SUMMARY OF USING WILDFLY SWARM WITH CAMEL FOR MICROSERVICES

WildFly Swarm is a light-weight *just enough* application server which provides support for all of the JEE stack. This is very appealing for users whom are already using JEE. Users not using JEE can of course 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 then you can find yourself in situations when using a newer version of Apache Camel where there hasn't been a WildFly Swarm Camel release yet, and thus you are stuck on using the older version.

WildFly Swarm is not the only just enough server on the market. Spring Boot is another very popular choice which works very well with Camel.

7.2.4 Spring Boot with Camel as microservice

Spring Boot is an opinionated framework for building microservices with minimal fuss that *just works* (famous last word). Spring Boot is designed with convention over configuration and allows developers to get started quickly to develop microservices with reduced boilerplate, configuration and fuss. Spring Boot does this by:

- Auto configuration and much reduced configuration needed
- Providing curated list of starter dependencies
- Simplified application packaging as a standalone fat jar
- Optional application information, insights, and metrics

Lets get started with Spring Boot and try first without Camel, and then followed by adding Camel to an existing Spring Boot application.

GETTING STARTED WITH SPRING BOOT

We will be using the Spring Boot starter website (<http://start.spring.io>) to get started. As dependencies we chose only Web to start with a plain web application. Figure 7.2 shows where we are going.

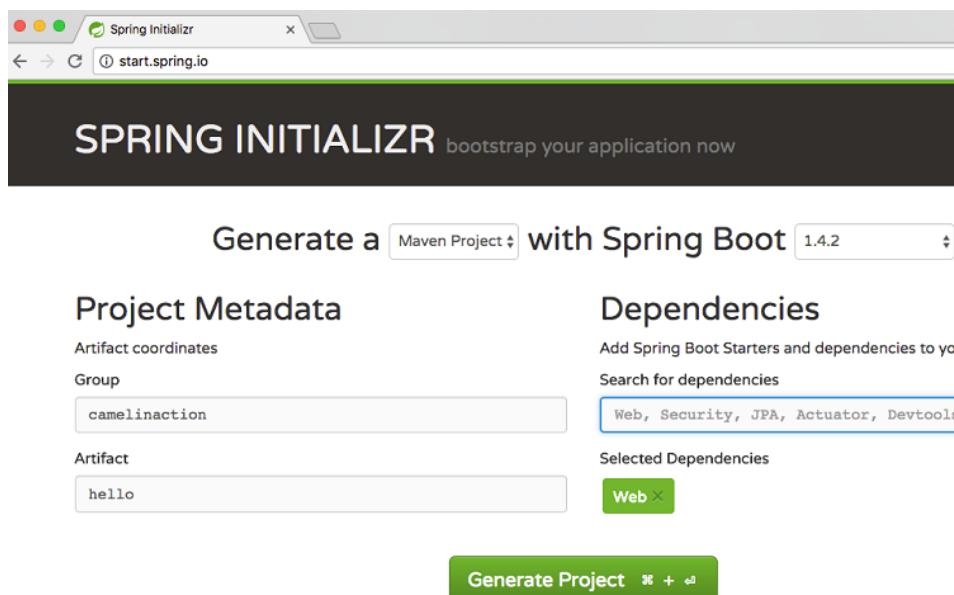


Figure 7.2 - Create a new Spring Boot project from the Spring starter website. We have only selected Web as dependencies to start with a plain web application.

The project is downloaded when clicking the Generate Project button. We then unzip the downloaded source code and are ready to run the Spring Boot application using Maven:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

```
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 will learn how to use the Camel Maven archetypes in the following 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 don't do anything yet so lets add a REST endpoint to return a hello message.

ADDING REST TO SPRING BOOT APPLICATION

We want to add a REST endpoint which returns a simple hello message. Spring provides REST support out of the box which you can build using Java code as shown in listing 7.6.

Listing 7.6 - RestController exposes a REST endpoint using Spring Rest

```
package camelinaction;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/spring")
public class HelloRestController {

    @RequestMapping(method = RequestMethod.GET, value = "/hello",
                    produces = "text/plain") ③
    public String hello() {
        return "Hello from Spring Boot";
    }
}
```

- ① Define this class as a REST endpoint
- ② Root mapping for all requests
- ③ HTTP GET service mapped to /hello

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 ③ at the method exposes a REST service under `/spring/hello` as a HTTP GET service which produces plain text content.

The source code of the book carries this example in the `chapter7/springboot` directory which you can try using the following Maven goal:

```
mvn spring-boot:run
```

And then from a web browser open `http://localhost:8080/spring/hello`

In the sections to follow we will show you two ways of adding Camel to Spring Boot. First we will add Camel to the existing `HelloRestController` class then followed by adding 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 use the template from the controller methods as shown in listing 7.7.

Listing 7.7 - Adding Camel to Spring controller class

```
@RestController
@RequestMapping("/spring")
public class HelloRestController {

    @EndpointInject(uri = "geocoder:address:current") ①
    private FluentProducerTemplate producer;

    @RequestMapping(method = RequestMethod.GET, value = "/hello",
                    produces = "text/plain")
    public String hello() {
        String where = producer.request(String.class); ②
        return "Hello from Spring Boot and Camel. We are at: " + where;
    }
}
```

- ① Dependency injecting Camel `FluentProducerTemplate` to controller class
- ② Using the template to request the endpoint and receive the response as a `String`

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 known 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 level of degree - my current location was 23km off by the geocoder). The controller then uses the injected Camel template to contact the geocoder when the `hello` methods 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 which is 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 how the artifactId of the Camel components uses the suffix `-starter`. For every Camel component there is a corresponding Camel Spring Boot curated starter component which is recommended to use. The starter component has been specially adjusted to work with Spring Boot and also includes support for Spring Boot auto configuration which we will cover later in this section.

The source code of the book contains this example in `chapter7/springboot-rest-camel` which you can try using the following goal

```
mvn springboot:run
```

And then from a web browser open `http://localhost:8080/spring/hello`

This example does not use any Camel routes. Lets take a look at how you can build this example using a Camel route instead of a Spring controller.

USING CAMEL ROUTES WITH SPRING BOOT

Camel routes are very natural to Camel applications and they are of course also supported in Spring Boot. You can write your Camel routes in both Java or XML DSL, however as Spring Boot is Java centric you may take on using Java DSL instead of XML. Lets rewrite the previous example that uses a Spring controller to expose a REST service to use a Camel route instead. Listing 7.8 shows how this can be done:

Listing 7.8 - Use Camel route in Spring Boot to expose a REST service

```
@Component
public class HelloRoute extends RouteBuilder {  

    ①  

    @Bean
    ServletRegistrationBean camelServlet() {  

        ②
        ServletRegistrationBean mapping = new ServletRegistrationBean();
        mapping.setName("CamelServlet");
        mapping.setLoadOnStartup(1);
        mapping.setServlet(new CamelHttpTransportServlet());
        ③
        mapping.addUrlMappings("/camel/*");
        return mapping;
    }  

    ④  

    @Override
    public void configure() throws Exception {
        rest("/")
            .get("hello")
            .to("direct:hello");
```

```

from("direct:hello")
    .to("geocoder:address:current")
    .transform().simple("Hello. We are at: ${body}");
}
}

```

- ➊ @Component annotation to let Camel route be auto discovered
- ➋ Register Camel servlet to use for REST service
- ➌ Camel HTTP servlet component
- ➍ Camel rest-dsl to define a REST service in the route
- ➎ Camel route

When creating Camel routes in Java DSL with Spring Boot you should annotate the class with `@Component` which ensures the route is auto detected by Spring and automatic added to Camel when the application starts up. Because we want to expose a rest service from a Camel route we need to use a HTTP server component. Spring Boot comes with a servlet engine out of the box, and therefore we need to configure a servlet to be used. In Spring Boot you do this by the `ServletRegistrationBean` which is configured from the Java method ➋ where you can set the actual servlet to be used ➌.

TIP It is expected that configuring the Camel servlet bean ➋ becomes much easier from Camel 2.19 onwards.

In Camel you can define rest services using a DSL known as `rest-dsl` ➍ which will use the servlet component that has just been configured. The Camel route ➎ then comes next and notice how the `rest-dsl` calls the route using the `direct` endpoint, which is how you can link them together.

NOTE We will cover the `rest-dsl` extensively in chapter 10. However we though the three lines of code would not knock you down.

We have of course provided the source code for this example with the book which you can try from the `chapter7/springboot-camel` directory using the following Maven goal:

```
mvn spring-boot:run
```

And 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 such as shown below:

```
@SpringBootApplication
@ImportResource("classpath:mycamel.xml")
```

```
public class SpringbootApplication {
```

The XML file is a regular Spring <beans> XML which can contain <bean>'s and <camelContext> as shown below:

```
<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 whom either prefer Camel XML DSL or whom are migrating existing Spring applications which has been configured with Spring XML files to run on Spring Boot.

We have provided an example with the source code in chapter7/springboot-xml which you can try using the following Maven goal:

```
mvn spring-boot:run
```

The perceptive eye may have noticed this example are using 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 which Camel can use.

In the previous example we are using Camel property placeholder in the <log> EIP:

```
<log message="Spring Boot says {{hello}} to me"/>
```

We then define a property with key hello in the application.properties:

```
hello=I was here
```

And hey presto no surprise Camel is able to lookup and use that value without you having to configure anything.

Spring Boot allows to override property values in a number of ways. One of them is by configuring JVM system properties. So instead of using hello=I was here, we can override this when we 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
```

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 microservices where one of them was being highly configurable.

Another characteristics of a microservice is that it should be self monitoring.

MONITORING CAMEL WITH SPRING BOOT

Spring Boot provides what is called actuator endpoints to monitor and interact with your application. Spring Boot includes a number of built-in endpoints which can report many kind of information about the application.

To enable Spring Boot actuator you would need to add its dependency to the Maven pom.xml file as shown:

```
<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 of course Camel has. This means your Camel applications can provide health indicators if the Camel application is healthy or not.

You can try this by running the chapter7/springboot-rest-camel example by

```
mvn spring-boot:run
```

And then from a web browser open: <http://localhost:8080/health>

By always using /health endpoint for your Spring Boot applications you can more easily setup centralized monitoring to use this endpoint to query the health information about your running applications.

We have a few words to say about why Camel and Spring Boot works well together for developing microservices applications.

SUMMARY OF USING CAMEL WITH SPRING BOOT FOR MICROSERVICES

Spring Boot is a very popular *just enough* application server 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 its a great choice. However if you are using or coming from a JEE background then we would recommend to look at WildFly Swarm instead.

The last microservices container we will look at is called vert.x.

7.2.5 Using Vert.x to build microservices

On the Eclipse Vert.X website the project describes itself as *Vert.x is a toolkit for building reactive applications on the JVM*. There are three important points in this description.

As a toolkit Vert.x is not an application server, a container or a framework. Vert.x is just a JAR file (`vertx-core`), so a Vert.x application is an application that uses this jar file.

Secondly 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 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 message passing between its components. This establishes a boundary between components that ensures loose coupling, isolation and location transparency.

Finally Vert.x applications runs on the JVM which means Vert.X applications can be developed using any the JVM languages such as Java, Groovy, Scala, Kotlin, Ceylon, JavaScript etc. The polyglot nature of Vert.x allows you to use the most appropriate language for the task.

TIP Clement Escoffier wrote an excellent Vert.x tutorial which is recommended to read to get up to speed about Vert.x: <http://escoffier.me/vertx-hol>

The scope of this book is to cover all about Apache Camel. However we want to give room for Vert.X in this book as we think its an awesome toolkit that has great potential together with Camel.

BUILDING A DONG SIMULATOR USING VERT.X

What follows next is a true event that happened in the life of Claus Ibsen (the author). I am hosting a x-mas party with seven of my old friends in the second weekend of December 2017.

When we were younger we would have these football weekends, where we would watch english football. It was back in the late 90's and up to mid 00's. During a football weekend the state lottery company would issue a football pools coupon with 13 games. One of these games was the TV game and out of the remainder 12 games each of us would select a game. We would then watch the TV game and the other games was played at the same time. Whenever a goal was scored it would be announced in the TV with a *dong* sound - and hence why it was known as dong bold (bold = football). The 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).

Fast forward to today. Now the football games are much more scattered during a game week and its hard to find a number of games play at the exact same time. So I wanted to build a goal simulator and use a good old classic game from the 90's or 00's. However it was harder to find a full length football game on the internet going back so many years. However I managed to find a great local derby between Manchester United and City from February 2004. As a bonus the sides have players in the lineup which we remember such as Ryan Giggs, Rio

Ferdinand, Gary Neville, Paul Scholes, Christian Ronald, Ole Gunnar Solskjaer, and Ruud van Nistelrooy. Today I can hardly remember anyone from the Manchester United side besides Zlatan Ibrahimovic.

I then researched and found which other games were playing on that same day and build my own football pools coupon with nine games; one TV game and a game for each of the eight of us. I then played the game in its entirety with the goal simulator running which then would flash and play the dong sound for each goal. This was then recorded which allowed me to easily playback the video clip during the party. Figure 7.3 shows a screenshot of the game in action with the goal simulator on the right hand side.



Figure 7.3 - Dong simulator playing the TV game with live goal scorer updates on the right hand side.

We had a great weekend and playing dong again was a blast. Manchester United won 4-2 with goals from Scholes, 2 x Nistelrooy and Ronaldo.

That was the fun part, lets talk about how to implement this using Vert.X.

THE ARCHITECTURE

Figure 7.4 illustrates the key components in the architecture.

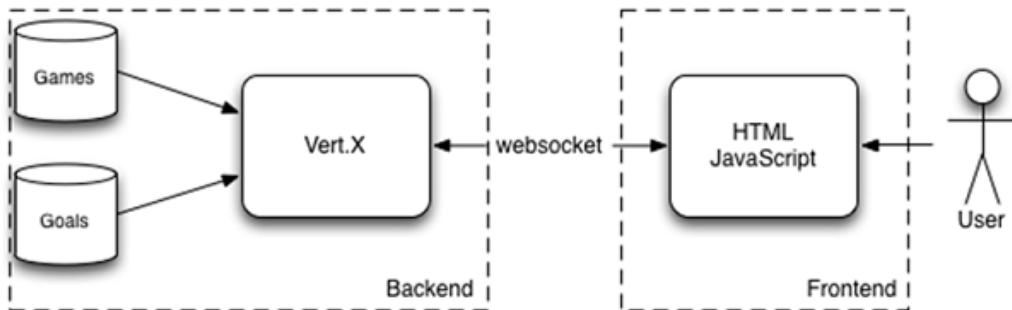


Figure 7.4 - In the backend the Vert.X applications loads the games and goals from disk. The frontend is a HTML web application with embedded JavaScript which uses websocket to communicate with the Vert.X backend.

The backend is implemented as a Vert.X application using Java. Information about the games and the goal scorers are stored in plain CSV text files which is loaded into the Vert.X application upon startup. The frontend is a plain HTML file with embedded JavaScript. The JavaScript uses a Vert.X JavaScript client which handles all communication to the backend using websocket. The entire application is packed together as a single fat jar, which can run as a Java application on the JVM.

The source code of the book contains this example in the chapter7/vertx directory. You can try this by running the following Maven goal:

```
mvn compile vertx:run
```

... and then open a web browser on <http://localhost:8080>

The application is built using Java and HTML code which the following intriguing parts.

THE JAVA CODE

You develop your Vert.X application in Java code in what is called a verticle which are deployed and running in the Vert.X instance. The verticle is just an abstract class which has start and stop methods and easy access to the vert.x instance itself.

Listing 7.9 shows the verticle for the dong simulator.

Listing 7.9 - Vert.X verticle for the dong simulator

```
import io.vertx.core.AbstractVerticle;

public class LiveScoreVerticle extends AbstractVerticle {          ①

    public void start() throws Exception {
        Router router = Router.router(vertx);                      ②

        BridgeOptions options = new BridgeOptions()
            .addInboundPermitted(new PermittedOptions().setAddress("control"))
            .addOutboundPermitted(new PermittedOptions().setAddress("clock"))
            .addOutboundPermitted(new PermittedOptions().setAddress("games")) ③
    }
}
```

```

    .addOutboundPermitted(new PermittedOptions().setAddress("goals"));

    router.route("/eventbus/*")
        .handler(SockJSHandler.create(vertx).bridge(options, event -> {
            if (event.type() == BridgeEventType.SOCKET_CREATED) {
                vertx.setTimer(500, h -> initGames());           ④
            }
            event.complete(true);
        }));
    router.route().handler(StaticHandler.create());          ⑥

    vertx.createHttpServer()
        .requestHandler(router::accept).listen(8080);        ⑦

    initControls();                                       ⑧
    streamLiveScore();                                    ⑧
}

```

- ① The verticle is extending AbstractVerticle class
- ② Create Vert.X router to setup websocket and HTTP server
- ③ Allowed inbound and outbound traffic on eventbus
- ④ Route websocket to Vert.X eventbus
- ⑤ A new websocket client connected so initialize list of games to client
- ⑥ Add static HTML resources to Vert.X router
- ⑦ Vert.X router listen on port 8080
- ⑧ Initialize game controls and start live score streams

The dong simulator code from listing 7.9 shows how to build a Vert.X application as a verticle. The `LiveScoreVerticle` class extends `io.vertx.core.AbstractVerticle` ①. In the start method we have the necessary code to startup, such as creating a Vert.X router for HTTP and websocket ②. Then we setup allowed inbound and outbound communication ③ to the router with the following four eventbus addresses: control, clock, games and goals. The communication between the backend and frontend is using websocket which we add to the router ④. Whenever a new client is connected the `SOCKET_CREATED` event is emitted to the backend 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 lastly the game controls and the goal score stream is started ⑧.

The `LiveScoreVerticle` class has more code to initialize the game list, react upon game control buttons pushed, game clock advances and the actual stream of goals. For example the list of games is initialized as shown in listing 7.10.

Listing 7.10 - Initializing the list of games

```

private void initGames() {
    try {
        InputStream is = LiveScoreVerticle.class.getClassLoader()
            .getResourceAsStream("games.csv");
        String text = IOHelper.loadText(is);           ①
        Stream<String> games = Arrays.stream(text.split("\n"));
        games.forEach(game -> vertx.eventBus().publish("games", game)); ②
    }
}

```

```

} catch (Exception e) {
    System.out.println("Error reading games.csv file");
}

if (clockRunning.get()) {
    vertx.eventBus().publish("clock", "" + gameTime.get());      ③
} else {
    vertx.eventBus().publish("clock", "Stopped");                ③
}
}

```

- ① Load list of games from classpath
- ② Publish each game to the eventbus
- ④ Publish game clock time

The list of games is stored in a CSV file which is loaded ① . Each line in the CSV file is a game which gets published to the Vert.X eventbus at the games address ② . In addition the game clock state is published as well ③ .

As you can see from listing 7.10 its very easy to send messages to the Vert.X eventbus 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 dong simulator has buttons in the front end which controls 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 frontend to the backend using websocket on the Vert.X eventbus. The following code is all it takes in the backend to setup the consumer:

```

private void initControls() {
    vertx.eventBus().localConsumer("control", h -> {           ①
        String action = (String) h.body();                      ②
        if ("start".equals(action)) {
            clockRunning.set(true);                            ③
            vertx.eventBus().publish("clock", "" + gameTime.get()); ④
        } else if ("stop".equals(action)) {
            clockRunning.set(false);
            vertx.eventBus().publish("clock", "Stopped");       ③
        }
    });
}

```

- ① Setup consumer on the Vert.X eventbus control address
- ② The message body contains what action to perform
- ③ Either start or stop the game clock
- ④ Publish the new game clock state back to the frontend

From the Vert.X eventbus we setup a local consumer to listen on address control ① and upon each message received it 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 we react accordingly to either start or stop the

game clock ③ . We then publish the new state back so the web page can react and update its display ④ .

TIP We are using localConsumer which is for non-clustered Vert.X instances. If clustering is used then use consumer instead.

This was the key points from the Java code, lets switch over to the wild west of front end programming with web frameworks and JavaScript. Oh well its actually fairly simple, certainly with the Vert.X JavaScript client.

THE HTML CODE

Vert.X allows to embed web resources such as HTML and JavaScript files in the src/main/resources/webroot folder. We have the following files in this folder:

```
└── dong.m4r
└── index.html
└── vertx-eventbus.js
```

The dong.m4r file is the dong audio which is played when a goal is scored. The index.html file is the HTML file which we will 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 setup 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> ①
  <script src="//cdn.jsdelivr.net/sockjs/0.3.4/sockjs.min.js"></script> ②
  <script src="vertx-eventbus.js"></script> ③
</head>
```

In this example we are using the popular JQuery JavaScript library ① . For websocket communication Vert.X uses SockJS ② which provides fallbacks to a simulated websocket communication if the web browser does not support native websocket. The last script is to include Vert.X itself ③ .

In the <script> section we setup the frontend to connect to the Vert.X eventbus as shown in listing 7.11.

Listing 7.11 - Using Vert.X in the frontend to handle events from the eventbus

```
<script>
  var eb = new EventBus("http://localhost:8080/eventbus"); ①

  eb.onopen = function () {

    eb.registerHandler("clock", function (err, msg) {
      document.getElementById("clock").innerHTML = msg.body; ②
    });
  };

```

```

eb.registerHandler("games", function (err, msg) {           ③
    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
    if (msg.body === 'empty') {
        clearScorer();
        return;
    }

    playsound();

    // more code here not shown
})
};

```

- ① Connect to Vert.X eventbus on localhost:8080
- ② React when the game clock changes
- ③ React when the game list is updated
- ④ React when a goal is scored

To use Vert.X from JavaScript you need to create a new eventbus with the URL to the backend ① . Then the `onopen` method allows you to register handlers which react when messages are sent to eventbus addresses. In this example there are three addresses the frontend uses. When the game clock is updated ② . When the list of games is initialized ③ and of course when a goal is scored ④ .

For example when the game clock is updated ② then the frontend updates the HTML page by setting the `innerHTML` to the message body. The game clock is a HTML `<div>` element as shown below:

```
<div id="clock" class="clock"></div>
```

The source code in listing 7.11 has been abbreviated to not show the JavaScript code that manipulates the HTML elements to update the website. This is regular JavaScript and HTML code which the authors of this book are not ninja master, and therefore don't want to show our embarrassing skills in print. You are of course very much welcome to take a look at the source code, and any CSS and HTML ninjas are welcome to teach us how this can be styled and done better.

The source code is located in `chapter7/vertx` directory. You can try running the example using the following Maven goal:

```
mvn vertx:run
```

.. and then from a web browser open `http://localhost:8080`.

The first goal is scored in the 3rd minute so you have to wait a little before some action happens. But that is just enough time to fetch a beer from the fridge and be ready to drink when the goal is scored.

If you have been sitting back and relaxed for a while and had a few drinks as the goals pouring in, you may have been enlightened and noticed the goal simulator is not using Camel at all. Vert.X is surely a very awesome toolkit for building small reactive microservices. But this book is titled Camel in Action so lets update the simulator to use Camel together with Vert.X.

7.2.6 Using Camel together with Vert.X

Vert.X and Camel are both small and light-weight toolkits that work very well together. Figure 7.5 illustrates this principle with Vert.X and Camel working together in the same Vert.X application.

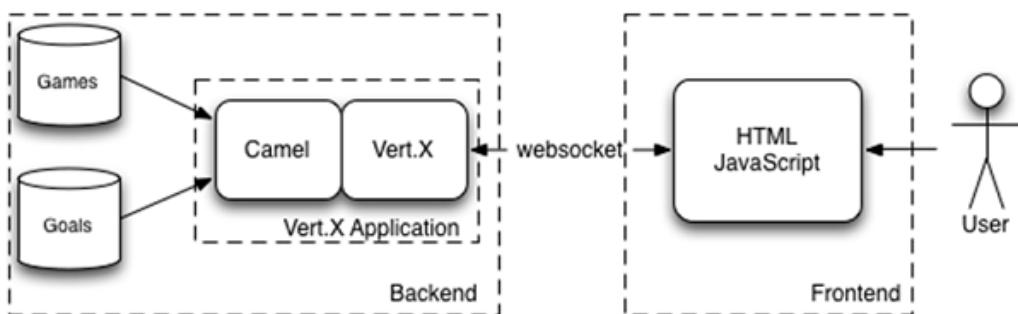


Figure 7.5 - Camel and Vert.X working together in the same Vert.X application in the backend. The other parts of the architecture are unchanged.

To use Camel with Vert.X you need to add camel-core and camel-vertx dependencies to the Maven pom.xml file:

```

<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-core</artifactId>
    <version>2.18.1</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-vertx</artifactId>
    <version>2.18.1</version>
</dependency>

```

In the Vert.X application you add Camel to the verticle class as shown in listing 7.12.

Listing 7.12 - 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
        camelContext.addRoutes(new LiveScoreRouteBuilder(vertx)); ②
        template = camelContext.createFluentProducerTemplate(); ③
        camelContext.start(); ④

        Router router = Router.router(vertx);
        ...
    } ⑤

    public void stop() throws Exception {
        template.stop();
        camelContext.stop(); ⑥
    }
}
```

- ① Create CamelContext
- ② Add Camel routes which uses the Vert.X instance
- ③ Create ProducerTemplate
- ④ Start Camel
- ⑤ Setup Vert.X Router
- ⑥ Stop Camel

The code to add Camel should be familiar to you. At first a CamelContext is created ② then routes is added ②. Then we create a ProducerTemplate ③ which will be used by Vert.X to trigger a Camel route. Camel is then started ④ and the following code is setting up Vert.X router ⑤ and what else which is similar to the previous example from listing 7.9. When the Vert.X application is stopping then we must remember to stop Camel ⑥ as well.

CALLING CAMEL FROM VERT.X

When a new client connects to the backend Vert.X will trigger the `SOCKET_CREATED` event which we use to obtain the list of games and send to the frontend so the website can be updated accordingly. In the previous example we used Java code to load the game list from a CSV file and send the information using Vert.X. This time we are using Camel, so the `SOCKET_CREATED` event is using the `ProducerTemplate` to trigger a Camel route by sending an empty message to the `direct:init-game` endpoint ① as shown:

```
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()); ①
        }
        event.complete(true);
    }));
}
```

① Calling Camel route using the ProducerTemplate

As you can see calling Camel from Vert.X is very simple, you just 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 which is used to route messages to/from the Vert.X eventbus and Camel. Listing 7.13 shows how this is done.

Listing 7.13 - Using Camel routes to stream live goal scores to Vert.X eventbus

```
public class LiveScoreRouteBuilder extends RouteBuilder {

    private final Vertx vertx;

    public LiveScoreRouteBuilder(Vertx vertx) {
        this.vertx = vertx;                                ①
    }

    public void configure() throws Exception {
        getContext()                                     ②
            .getComponent("vertx", VertxComponent.class)
            .setVertx(vertx);

        from("direct:init-games").routeId("init-games")   ③
            .log("Init games event")
            .to("goal:games.csv")                           ④
            .split(body())
            .to("vertx:games");                            ⑤

        from("goal:goals.csv").routeId("livescore").autoStartup(false) ⑥
            .log("Goal event: ${header.action} -> ${body}")
            .choice()
                .when(header("action").isEqualTo("clock"))
                    .to("vertx:clock")                      ⑦
                .when(header("action").isEqualTo("goal"))
                    .to("vertx:goals");                   ⑧

        from("vertx:control").routeId("control")           ⑨
            .log("Control event: ${body}")
            .toD("controlbus:route?routeId=livescore&async=true&action=${body}");
    }
}
```

- ① Inject Vert.X instance in constructor
- ② Setup Vert.X instance on Camel vertx component
- ③ Route to initialize game list
- ④ Get list of games from goal component
- ⑤ Split each game and send the vertx eventbus
- ⑥ Route to stream live goal scores
- ⑦ Update game clock
- ⑧ Update goal score

9 Route for control buttons

The `LiveScoreRouteBuilder` class is injected with the Vert.X instance ① in the constructor because we need to configure the Camel vertx component to use this instance ② .

Then follows three Camel routes. 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 list from the file system. The frontend except one message per game, and hence we need to split the game list before sending to the vertx eventbus on the games address ⑤ .

NOTE To hide the complexity of loading the game list and streaming live goal scores we have built a Camel component named `goal`.

The second route is responsible for streaming game clock and goal score updates ⑥ which is routed using a Content Based Router EIP to either the `clock` ⑦ or `goals` ⑧ address on the vertx eventbus. Pay attention to the fact that the route has been configured to not auto start. The reason is that we want the user to click the start button in the frontend, which is controlled by the last route ⑨ . The controlbus component is capable of starting and stopping routes. Notice how we refer to the `livescore` route using the `routeId` parameter on the controlbus endpoint.

TIP Chapter 16 will cover much more about the controlbus component with the topic 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 frontend 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 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.

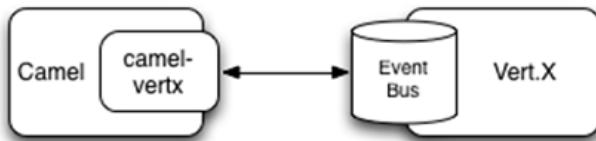


Figure 7.6 - Camel and Vert.X exchange messages using the Vert.X eventbus. Camel facilitates this using the camel-vertx component.

The only touch point between them are exchanging messages using the Vert.X eventbus using the camel-vertx component. Figure 7.6 illustrates this principle.

TRYING THE EXAMPLE

You can find the source code in the chapter7/vertx-camel directory which you can try using the following Maven goal:

```
mvn compile vertx:run
```

... and then from a web browser open <http://localhost:8080>.

Okay lets end the drinking game and conclude our Vert.X coverage in this book with some final words.

SUMMARY OF USING CAMEL WITH VERT.X FOR MICROSERVICES

Building the dong football simulator has been a fun ride with using Vert.X, and then later adding Camel to the mix. This kind of small application runs really well with Vert.X which has great support for HTML and JavaScript clients. Notice how easy it is to exchange data between the Java backend and the HTML frontend using the Vert.X eventbus. The web frontend is a modern (well except for bad looking CSS styling) HTML5 single page application which reacts to live stream of events. This is where Vert.X shines really well.

So how does this compare to Camel then? Well Vert.X is focused on smaller reactive applications, and Camel is focused on messaging and integration. Its the combination of the two that gives the $2 + 2 = 5$ synergy.

There is a lot of great to say about Vert.X and we recommend you take a look at the project and keep an eye on it for the future. Because Vert.X is reactive, asynchronous and non-blocking it puts the burden on the developer to understand its APIs really well. This takes time to master and grasp. Camel on the other hand hides a lot of that complexity and as a developer you can get very 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.

Vert.X has a lot of greatness but there is also some room for improvements. For example Vert.X does not come with a good story for configuration management, as opposed to Spring Boot for example. Vert.X neither comes with functionality for generic health check, which is a

critical feature for containerized applications which we will talk about in chapter 17. Vert.X also comes with very little logging so it can be harder to trouble shoot. Fat jars is a popular packing today and Vert.X does not provide its own Maven plugin to make this easy. But thanks to people in the community whom stepped up and created the vertx-maven-plugin (<https://vmp.fabric8.io>) which makes this easy to do.

Despite these setbacks we think Vert.X is an awesome project and worthwhile to keep an eye on and give it a try if you get the chance. Camel and Vert.X are awesome together.

Okay we just past the halfway mark in this thrilling chapter. Maybe take a break and come back to this chapter again with fresh energy because we are now changing focus.

No man is an island is a famous phrase from a four hundred year old poem. What we mean is that no microservice is alone. Microservices are composed together to conduct business transactions. The next topic is how you can build microservices with Camel that calls others services and the implications it brings to the table.

7.3 Calling other microservices

In the land of microservices each service is responsible for providing functionality to other collaborators. One of the microservices characteristic we discussed in the beginning of this chapter is that build distributed systems is hard and that microservices must be designed to deal with failures. In this section we will first cover how to build Camel based microservices that call other services (without design for failures) to cover the basics first. Then the following section we cover the EIP patterns which can be applied to design for failures.

THE STORY OF THE BUSY DEVELOPER

At Rider Auto Parts you have been dazzling a bit with microservices but haven't really kicked the tires yet due to a busy work schedule. Does it all sound to familiar? Yeah even authors of Camel books knows this feeling just too well. So to put yourself back on the saddle again you sent your wife and kids away to visit their in-laws and leaving you alone in the house for the entire weekend. The goal of the weekend is to enjoy an occasional beer while studying and writing some code to familiarize yourself more with how to build a set of microservices that collectively solves a business case. You don't want to settle down on a particular runtime and therefore want to implement the services using different technologies.

The business case you will attempt to implement using microservices is illustrated in figure 7.7.

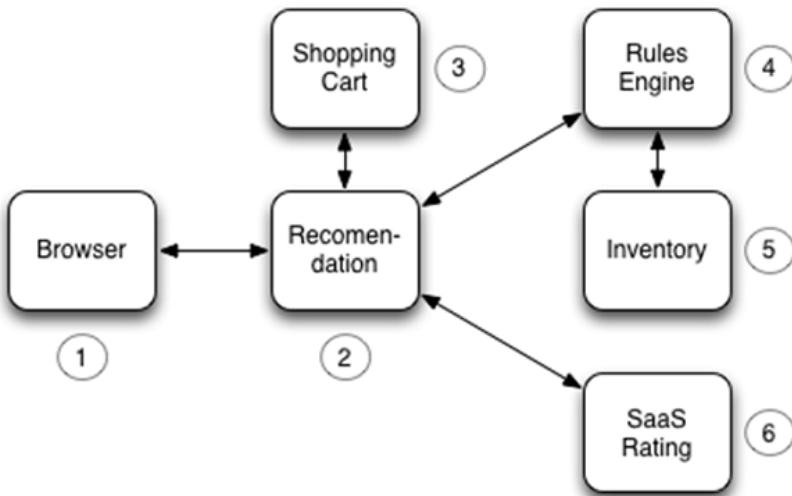


Figure 7.7 - Rider Auto Parts recommendation system. From the web browser (1) the recommendation service (2) obtains items currently in the shopping cart (3) and then calls the rules engine (4) to compute items to be recommended, which takes into account number of items currently on stock (5). Finally 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 decided to have a jump start by implementing a prototype. Users whom 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 is fed into the rules engine ④ which is used to rank recommendations. The inventory system ⑤ in the backend hosts information about what items are currently in stock and thus can help rank items which can be shipped to customers immediately. An external SaaS service is used to help rank popular items based on a world wide user based rating system ⑥.

TECHNOLOGIES

The technology stack for the recommendation system is illustrated in figure 7.8.

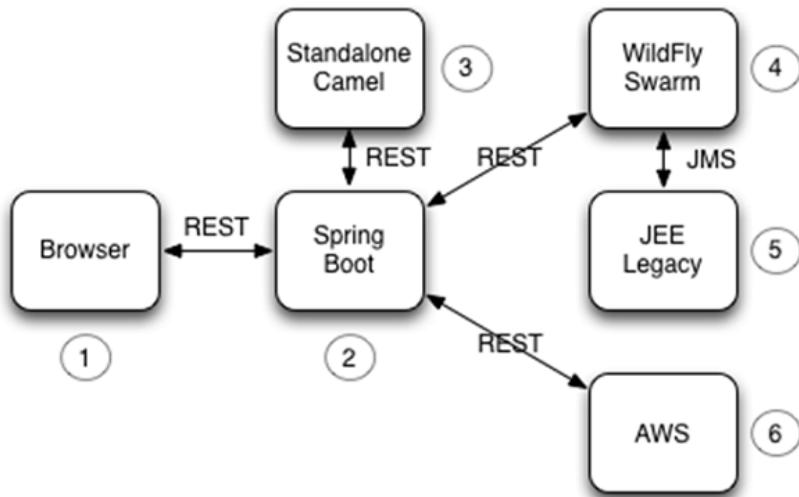


Figure 7.8 - Web browser ① communicates using HTTP/REST to the recommendation system ② running on Spring Boot. The shopping cart is running standalone Camel ③ . The rules engine ④ running inside WildFly Swarm is integrated using REST. The inventory system is hosted on a legacy JEE application server ⑤ which is integrated using JMS messaging. The rating system is hosted in the cloud on AWS ⑥ .

You only have one weekend so lets started hacking some code. You have decided to implement each microservice as a separate module in the source code. To keep things simple you decided to keep all the source in the same git repository. The source code structure is as shown:

```

prototype
├── cart           Shopping cart
├── inventory      Inventory system
├── recommend      Recommendation microservice
├── rules          Rules engine
└── rating         External SaaS rating system

```

You can find the source code for this example in the chapter7/prototype directory.

In the following section we will cover how you have built this prototype and highlight they 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 it has more moving parts than the services to follow.

The prototype consist 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
- SpringApplication.java - The main class to bootstrap this service
- application.properties - Configuration file

The *meat* of this service is the RecommendController class which is shown in listing 7.12.

Listing 7.12 - Recommendation REST service

```

@RestController                                     ①
@RequestMapping("/api")                         ②
@ConfigurationProperties(prefix = "recommend")
public class RecommendController {

    private String cartUrl;                      ③
    private String rulesUrl;                     ③
    private String ratingsUrl;                   ③

    private final RestTemplate restTemplate = new RestTemplate();      ④

    @RequestMapping(value = "recommend", method = RequestMethod.GET,
                     produces = "application/json")                  ⑤
    public List<ItemDto> recommend(HttpServletRequest session) {
        String id = session.getId();

        CartDto[] carts = restTemplate.getForObject(
            cartUrl, CartDto[].class, id);                           ⑥
        String cartIds = cartsToCommaString(carts);

        ItemDto[] items = restTemplate.getForObject(
            rulesUrl, ItemDto[].class, id, cartIds);                ⑦
        String itemIds = itemsToCommaString(items);

        RatingDto[] ratings = restTemplate.getForObject(
            ratingsUrl, RatingDto[].class, itemIds);               ⑧

        for (RatingDto rating : ratings) {
            appendRatingToItem(rating, items);
        }
        return Arrays.asList(items);
    }
}

```

- ① Define class as REST controller
- ② Inject configuration properties with prefix recommend.
- ③ Fields with injected configuration values
- ④ REST template used to call other microservices
- ⑤ Define REST service
- ⑥ Call shopping cart REST service
- ⑦ Call rules engine REST service
- ⑧ Call SaaS rating REST service

Spring Boot makes it easy to define REST service in Java by annotation the class with `@RestController` and `@RequestMapping` ① . In the chapter introduction we talked about the good practice to externalize your configuration. Spring Boot allows to automatic inject getter/setters ③ from its configuration file by annotation the class with `@ConfigurationProperties` ② . In this example we have 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/{itemIds}
```

To call other REST services we use Springs `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 the `RestTemplate`.

Automatic mapping JSON response to DTO

Notice how the `RestTemplate getForObject` method automatic 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 cannot use a `List` type due to Java type erasure in collections. For example the following cannot compile:

```
List<CartDto> carts = restTemplate.getForObject(cartUrl, List<CartDto.class>, id);
```

As you can see calling another microservice using Spring's `RestTemplate` is easy as its just one line of code with the `RestTemplate`. Have we seen this kind before? Yes of course Camel's `ProducerTemplate` or `FluentProducerTemplate` is also easy to use for sending messages to Camel endpoints with one line of code.

In listing 7.12 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 using Camel with different containers. This time you have 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 listing 7.13.

Listing 7.13 - Simple shopping cart implementation class

```
@ApplicationScoped
@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
    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
    Set<CartDto> dtos = content.get(sessionId);
    if (dtos != null) {
        dtos.remove(itemId);
    }
}

public Set<CartDto> getItems(@Header("sessionId") String sessionId) { ④
    Set<CartDto> answer = content.get(sessionId);
    if (answer == null) {
        answer = Collections.EMPTY_SET;
    }
    return answer;
}
}

```

- ① CDI application scoped bean named cart
- ② Method to add item to cart stored
- ③ Method to remove an item from the cart
- ④ Method to get the items from the cart

The `CartService` class has been annotated with `@ApplicationScoped` ① because only one instance is needed at runtime. The name of the instance can be assigned using `@Named`. The class implements three methods ② ③ ④ to add, remove, and get items from the cart. Notice how 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 will cover in a little while.

TIP We covered bean parameter bindings in chapter 4.

In listing 7.13 the shopping cart uses `CartDto`'s in the add and get methods ② ④ . This class is implemented as a plain POJO as shown below:

```

public class CartDto {
    private String itemId;
    private int amount;

    @ApiModelProperty(value = "Id of the item in the shopping cart") ①
    public String getItemId() {

```

```

        return itemId;
    }

    @ApiModelProperty(value = "How many items to purchase")
    public int getAmount() {
        return amount;
    }

    // setter methods omitted
}

```

①

① Swagger annotation to document the fields

The `CartDto` class has two fields to store the item id and the number of items to purchase. The getter methods ① has been annotated with Swagger's `ApiModelProperty` to include descriptions of the fields, which will be used in the API documentation of the microservice.

TIP We will cover REST services and Swagger API documentation in chapter 10

Earlier in this chapter, in code listing 7.8, you had a quick glimpse of Camel's REST DSL. Now embrace yourself because this time we dive deeper and you implemented the shopping cart microservice using Camel's REST DSL and with automatic API documentation using Swagger. All this is done from the following Camel route as show in listing 7.14.

Listing 7.14 - Shopping Cart microservice using Camel's REST DSL

```

@Singleton
public class CartRoute extends RouteBuilder {

    public void configure() throws Exception {

        restConfiguration("jetty").port("{{port}}").contextPath("api")           ①
            .bindingMode(RestBindingMode.json)                                     ②
            .dataFormatProperty("disableFeatures", "FAIL_ON_EMPTY_BEANS")       ③
            .apiContextPath("api-doc")                                            ④
            .enableCORS(true);

        rest("/cart").consumes("application/json").produces("application/json")
            .get()                                                               ⑤
                .outTypeList(CartDto.class)
                .description("Returns the items currently in the shopping cart")
                .to("bean:cart?method=.getItems")                                    ⑥
            .post()                                                               ⑦
                .type(CartDto.class)
                .description("Adds the item to the shopping cart")
                .to("bean:cart?method=addItem")
            .delete().description("Removes the item from the shopping cart")   ⑧
                .param().name("itemId")
                    .description("Id of item to remove")
                .endParam()
                .to("bean:cart?method=removeItem");                                 ⑨
    }
}

```

- 1 Configure REST-DSL to use Jetty component
- 2 Turn on JSON binding to/from POJO classes
- 3 Turn off binding error on empty lists/beans
- 4 Enable Swagger API documentation
- 5 GET /cart service
- 6 POST /cart service
- 7 DELETE /cart service
- 8 Call the CartService bean methods

The REST service is using Came's jetty component ① as the HTTP server. To work with JSON from Java POJO classes we have 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 expose three REST services ⑤ ⑥ ⑦ each calling the `CartService` bean ⑧ . Notice how each of the REST services documents their input and output types and parameters which becomes part of the Swagger API documentation.

RUNNING THE SHOPPING CART SERVICE

The source code for the shopping cart microservice is located in chapter7/prototype/cart directory which you can start 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 as an extension to your web browser which has a built-in REST client as shown in figure 7.9.

The screenshot shows the Postman application interface. The top navigation bar includes 'Runner', 'Import', 'Builder' (which is selected), 'Team Library', 'Sync Off', and 'Sign In'. The left sidebar has sections for 'History' (containing a 'GET' entry for http://localhost:8282/api/cart?sessionId=123) and 'Collections'. The main workspace shows a 'POST' request to 'http://localhost:8282/api/cart?sessionId=123'. The 'Body' tab is selected, showing a JSON payload:

```

1 {
2   "itemId": "444",
3   "amount": 2
4 }

```

Figure 7.9 - Using Postman to send HTTP POST request to 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 involves POST, PUT, and DELETE operations. Notice how 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 8888:8888 sjeandeaux/docker-swagger-ui
```

Then open a web browser on

```
http://localhost:8888/swagger-ui
```

.. and then type the following url in the url field and click Explore button

```
http://localhost:8282/api/api-doc
```

Then you should see the cart service which you can expand and try as shown in figure 7.10.

The screenshot shows the Swagger UI interface for the shopping cart service. At the top, there's a navigation bar with back, forward, and refresh buttons, a search icon, and a star icon. Below that is a green header bar with the word "swagger" and a "Explore" button. The main content area has a title "cart" and three tabs: "DELETE /cart", "GET /cart", and "POST /cart". The "POST /cart" tab is currently active. It shows the following details:

- DELETE /cart**: Description: Removes the item from the shopping cart.
- GET /cart**: Description: Returns the items currently in the shopping cart.
- POST /cart**: Description: Adds the item to the shopping cart.

Parameters

| Parameter | Value | Description | Parameter Type | Data Type |
|-------------|---|-------------|----------------|---|
| body | <pre>{ "itemId": "555", "amount": 3 }</pre> | | body | Model Example Value <pre>{ "itemId": "string", "amount": 0 }</pre> |

Below the table, it says "Parameter content type: application/json". At the bottom of the interface are two buttons: "Try it out!" and "Hide Response".

Figure 7.10 - Using Swagger UI to explore the shopping cart REST service. Here we are about to call the POST method to add an item to the shopping cart.

This was a glimpse of some of the powers Camel provides for REST services which we have devoted the entire chapter 10 to cover in much more depth. Lets move on to the next microservice you developing during the weekend, which is the rules and inventory services.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

7.3.3 Rules and Inventory prototypes

Not all microservices are using REST services. For example the Rider Auto Parts inventory system is only accessible using a JMS messaging broker and XML as data format. This means the rules microservice needs to use JMS and XML and therefore you decided to use a JEE micro container (WildFly Swarm) to host the rules microservice. This is not bad news because a goal of the prototype is to use a variety of technologies to learn what works and what doesn't work and so on.

The rules microservice depends upon the inventory backend so lets start there first.

THE INVENTORY BACKEND

Because you develop the prototype from home then you don't want to access the Rider Auto Parts test environment and therefore built a quick simulated inventory backend using standalone Camel with embedded ActiveMQ broker which starts up using a main class. The embedded ActiveMQ broker listen for connections on its default port

```
<transportConnector name="tcp" uri="tcp://0.0.0.0:61616"/>
```

This means the rules microservice can communicate using JMS to the embedded broker using localhost:61616.

The backend 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
    <to uri="bean:inventory"/>                      ②
  </route>
</camelContext>
```

- ① JMS request/reply to query inventory
- ② Bean which simulates inventory backend

The rules microservice will send 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 is using WildFly Swarm as the container with embedded Camel. The service exposes a REST service that the recommendation service will use. Because WildFly Swarm is a JEE server you can implement the REST service using JAX-RS technology.

This is done by creating a JAX-RS application class:

```
@ApplicationPath("/")
public class RulesApplication extends Application {
```

①

- ① JAX-RS application using root (/) as context-path

This implementation is very short as we do need any custom configuration. The `@ApplicationPath` annotation ① is used to define the starting context-path which the REST services will use. The REST service is implemented in the `RulesController` class as shown in listing 7.15.

Listing 7.15 - JAX-RS REST implementation

```

@ApplicationScoped
@Path("/api")
public class RulesController {

    @Inject
    @Uri("direct:inventory")
    private FluentProducerTemplate producer;          ③

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/rules/{cartIds}")
    public List<ItemDto> rules(@PathParam("cartIds") String cartIds) { ④
        List<ItemDto> answer = new ArrayList<>();

        ItemsDto inventory = producer.request(ItemsDto.class);           ⑤

        for (ItemDto item : inventory.getItems()) {                      ⑥
            boolean duplicate = cartIds != null
                && cartIds.contains("" + item.getItemNo());
            if (!duplicate) {
                answer.add(item);
            }
        }

        Collections.sort(answer, new ItemSorter());                      ⑦
        return answer;
    }
}

```

- ① CDI application scope bean
- ② JAX-RS REST service using /api as context-path
- ③ Inject Camel's ProducerTemplate
- ④ HTTP GET service
- ⑤ Call Camel route to get items from backend inventory
- ⑥ Filter duplicate items
- ⑦ Sort and rank items to be returned

The controller class can be automatically discovered using CDI by annotating the class with `@ApplicationScoped` ①. If this is not done, then 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 which are serviced from the given /api context path. To call the inventory backend you will use Camel and therefore inject a `FluentProducerTemplate` ③ to make it easy to call a Camel route from within the class.

The rules method ④ is annotated with GET to setup this as a HTTP GET service returning data in JSON format. Notice how `@Path` and `@PathParam` maps to the `cartIds` method

parameter from the HTTP url using the `{cartIds}` syntax. For example if the REST service is called using:

```
HTTP GET /rules/123,456
```

... then `partIds` is mapped to the String value "123,456".

The injected `FluentProducerTemplate` ③ is used to let Camel call the backend service in one line of code ⑤. Because the service is returning a response you use the `request` method (`request = InOut, send = InOnly`) and automatic convert the response to `ItemsDto` POJO type.

The rules microservice then filters out duplicate items which is not intended to be returned ⑥ and finally the items are sorted using the `ItemSorter` implementation ⑦ that ranks the items to custom rules.

Camel is used to call the backend inventory service using JMS messaging as shown in the Camel route:

```
from("direct:inventory")
    .to("jms:queue:inventory")
    .unmarshal().jaxb("camelaction");
```

①
②

- ① Call the inventory backend using JMS
- ② Convert response from XML to POJO classes using JAXB

The route is using the Camel JMS component ① which needs to be configured. This is easily done in Java code by using CDI `@Produces` as shown:

```
@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. This can be done using POJO classes and JAXB. The Camel route converts the returned XML data from the backend to POJO 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 is all you need to do because WildFly Swarm will automatically convert the POJO classes to JSON using JAXB.

TIP When using JAXB then always remember to list the POJO classes in the `jaxb.index` file which is residing in the `src/main/resources` folder.

RUNNING THE RULES AND BACKEND SERVICES

You can run these services from the `chapter7/prototype` directory. First you need to start the backend service:

```
cd backend
mvn compile exec:java
```

... and then from another shell you can start the rules microservice:

```
cd rules
mvn wildfly-swarm:run
```

... and then from a web browser you can call the rules service using the following url:

```
http://localhost:8181/api/rules/123,456
```

The backend has been hardcoded to return three different items using ids 123, 456 and 789, therefore you can try urls such as:

```
http://localhost:8181/api/rules/123
http://localhost:8181/api/rules/456,789
```

There is only one microservice left in the prototype to develop which is the rating service.

7.3.4 Rating prototype

It has been a busy weekend and you have already built four prototypes and 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 which you can run locally. When it comes to the real deal then the rating service is hosted on Amazon AWS and Camel has the camel-aws component that offers integration with many of the Amazon technologies.

We have been using rest-dsl a few times already in this chapter, so you can quickly slash out the following code the setup a Camel REST service:

```
restConfiguration().bindingMode(RestBindingMode.json);           ①
rest("/ratings/{ids}").produces("application/json")           ②
    .get().to("bean:ratingService");
```

- ① Turn on JSON binding
- ② GET /ratings/{ids} REST service

The REST service is using JSON and therefore you turn on automatic JSON binding ① which allows Camel to bind between JSON data format and POJO classes. There is only one REST service that rates items ②. In interest of time you implemented the rating service to return random values as shown:

```
@Component("ratingService")
public class RatingService {                                     ①

    public List<RatingDto> ratings(@Header("ids") String items) { ②
        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));
    }
    return answer;
}
}

```

- ① Naming the bean ratingService
- ② Parameter mapping of rating ids
- ③ Generate random rating value

The class is annotated with `@Component("ratingService")` ① which allows Spring to discover the bean at runtime and allowing Camel to call the bean by its name, which is done from the Camel route using `to("bean:ratingService")`. The rating method ② maps to the rest-dsl `/ratings/{ids}` path by using the `@Header("ids")` annotation, which ensures 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 just a class with two fields:

```

public class RatingDto {
    private int itemNo;
    private int rating;
    // getter and setter omitted
}

```

Because you have turned on automatic JSON binding in the rest-dsl configuration then Camel will automatically convert from `List<RatingDto>` to JSON representation.

TIP We will revisit and learn much more about the rest-dsl in chapter 10.

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
```

... and then from a web browser you can open the following URL:

```
curl http://localhost:8383/api/ratings/123,456
```

Phew you have now implemented prototypes for all the microservices for the Rider Auto Parts recommendation system. So far you have been running each services in isolation its about time that you put them all together and see how they work together.

7.3.5 Putting all the microservices together

During a weekend you have 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 its time to put all pieces together and plug in the power and see what happens.

The accompanying source code of the book contains the entire prototype in the chapter7/prototype directory. You would need to open five command shells at once and for each shell run the following commands in the given order:

Start the shopping cart service

```
cd cart
mvn compile exec:java
```

Start the inventory service

```
cd inventory
mvn compile exec:java
```

Start the rules service

```
cd rating
mvn springb-boot:run
```

Start the recommendation service

```
cd rules
mvn wildfly-swarm:run
```

Start the rating service

```
cd recommend
mvn spring-boot:run
```

And then you can open a web browser and call the recommendation service using the following URL:

```
http://localhost:8080/api/recommend
```

And the response should contain three items being recommended. This maybe seem as no big deal but essentially you have five Java applications running isolated in their own JVM integrated together to implemented a business solution.

The weekend is coming to a close and you want to summarize what you have learned so far.

WHAT HAVE YOU LEARNED

First of all you have learned that it would be necessary 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 work day and how rapid and fast software is having a tremendous impact on business. Any companies are getting more and more worried about becoming *uberized*. As a forward thinker you take the matter in your hand and ensure to up your game and not 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 worry about Camel as it fit in any kind of container or runtime of your choosing. You learned Spring Boot is a great opinionated container for building microservices. The curated started dependencies is a snap to install and its auto configuration makes configuration consistent and more easy to learn and use. Camel works very well with Spring Boot and you have great power with testing as well (more to come in chapter 9). You dipped the toes with WildFly Swarm and learned a few things. It is a small and fast application server where you can easily include one the JEE parts you need. However WildFly Swarm does not yet have as good 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 cross over is fairly easy to learn.

There is a few things you want to do the family returns and your weekend comes to a close. In the beginning of this chapter you learned about some of the characteristics of microservices which you want to hold against what you have done this weekend. Your comments are marked in table 7.1.

Table 7.1 - Comments to microservices characteristics

| Characteristics | Comment |
|-----------------------------------|--|
| Small in size | Each microservice is surely small in size and does one thing and one thing only. |
| Handling transactional boundaries | Currently none of the microservices are transactional. Chapter 12 will cover transactions with Camel. |
| Self monitoring | Will be covered later in chapter 16 and 17. |
| Design for failure | An important topic which we haven't covered sufficient in this chapter. However there is more to come in chapter 11 covering error handling. |
| Highly configurable | Spring Boot is highly configurable. WildFly-Swarm does not yet offer configuration on the same level as Spring Boot. However you can leverage Camel properties component for Camel only configuration. |
| Smart endpoints and dumb routes | Can easily be done using Camel in smaller Camel routes. |
| Infrastructure automation | Will be covered later in chapter 17. |

You have come to realize that you haven't focused so much on the design for failure aspect. You still have the five microservices running so you quickly plays the devil advocate and stop the inventory service and then refresh the web browser. And kaboom the recommendation service fails with a HTTP error 500 and you find errors logged in the consoles. Oh boy how can you forget about this?

When building microservices or distributed applications it's paramount to assume failures happens and you must design with this in mind. How to design and deal with failures will be covered in depth in chapters 11 and 17. However we will not let you hanging in the dark, so lets take a moment to layout the landscape and show you how to use a design pattern to handle failures, and then return to this prototype before we reach the end of this chapter.

7.4 Designing for failures

Murph's Law stating *Whatever can go wrong, will go wrong* is very true in a distributed system such a microservice architecture. We could spend a lot of time on preventing errors from happening, but even so we cannot predict every case of how microservices can fail in a distributed system. Therefore we must face the music and design our microservices as resilient and fault tolerant.

In this section we will touch point on the following patterns you can use to deal with failures:

- Retry Pattern
- Circuit Breaker
- Bulk Head

The first pattern is the traditional solution to deal with failures - if something fails then try again. The second and third are patterns that has become popular with microservice architectures. These two patterns are often combined which you get with the Netflix Hystrix stack.

But we start from the top with the traditional way.

7.4.1 Using 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 it will succeed. The Camel error handler is using this pattern as its primary functionality. In this section we will only briefly touch the retry pattern and Camel's error handler because we have devoted the entire chapter 11 to cover all about this. However we can use the retry pattern in the rules prototype which contains 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")
            .unmarshal().jaxb("camelinaction");
    }
}
```

①

① Calling inventory microservice which may fail

To handle failures when calling the inventory microservice ① we can configure Camel's error handler to retry the operation as shown:

```
public void configure() throws Exception {
    errorHandler(defaultErrorHandler()
        .maximumRedeliveries(5)
        .redeliveryDelay(2000));①

    from("direct:inventory")
        .to("jms:queue:inventory")
        .unmarshal().jaxb("camelinaction");
}
```

① Configure error handler to retry up till 5 times

As you can see we have configured Camel's error handler to retry ① up till five times with two seconds delay between each attempt. So for example if the first two attempts fail, then the operation succeed at the third attempt and can continue route the message. Only if all attempts fails the entire operation fails, and an exception is propagated back the to caller from the Camel route.

If only the world was so easy however there are these five factors you must consider 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. However in every case it depends. For example a HTTP server may be unresponsive and retrying the operation in a bit may succeed. Likewise a database operation may fail due to a locking error due concurrent access to a table, which may succeed on next retry. But if the database fails due wrong credentials then retrying the operation will keep failing and therefor not a candidate for retries.

Camel's error handler supports configuring different strategies for different exceptions and therefore you can retry only network issues and not exceptions caused by invalid credentials. Chapter 11 covers all about Camel's error handler.

HOW OFTEN TO RETRY

You also have to consider how frequently you may retry an operation, and also the total duration. For example a real time service may only be allowed to retry an operation a few times with short delays before the service must respond its client. Opposed to a batch service which may be allowed to do many retries with longer delays.

The retry strategy should also consider other factors such as the SLAs of the service provider. For example if your call the service to 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 built in such mechanism otherwise consumers of the services can overload their system or degrade downstream services. Therefore the service

provider often provide information about the remaining request count allowed. So the retry strategy can read the returned request count and only attempt within the permitted parameters.

TIP Camel's error handler offers a retry-while functionality which allows to configure only to retry while a given predicate returns a true value.

IDEMPOTENCY

Another factor to consider is where calling a service is idempotent or not. A service is idempotent if calling the service again with the same input parameters yield the same result. A classic example is a banking service where calling the bank balance service is idempotent, whereas calling the withdrawal service is not idempotent.

Distributed systems are inherently more complex. Due to remote network a request may have been processed by the remote service but hasn't received back to the client, which then assume the operation failed, and retry the same operation, which then may cause unexpected side effects.

The service provider can use the Idempotent Consumer EIP pattern to guard its service with idempotency. We will cover this pattern in chapter 12, section 12.5.

MONITORING

Monitoring and tracking retries is important. For example if some operations have too many retries before they either fail or succeed then this indicate an area of problem, which 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 tracks every single message being routed at fine grained details. You will be able to pin point exactly which EIP or custom processor in your Camel applications that is causing retries or taking too long to process messages. You will learn all about monitoring your Camel applications in chapter 16.

TIP Camel uses the term redelivery when a message is retried.

TIMEOUT AND SLAS

When using the retry pattern, and retries kicks in, then the overall processing time increases by the additional time of every retry attempt. In a microservice architecture you may have SLAs and consumer timeouts which should be factored in. So 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 in the start of section 7.4.1 are using Camel's error handler for retries. What if you do not use Camel, how can you do retries?

7.4.2 Using Retry Pattern without Camel

So what if you do not use Camel, how can you use the retry pattern? You could use good old fashioned Java code with a try .. catch loop. For example in the recommendation service which is using Spring Boot and Java we could implement a retry pattern as shown in listing 7.16.

Listing 7.16 - Retry pattern using Java try .. catch loop

```
String itemIds = null;
ItemDto[] items = null;
int retry = 5 + 1;
for (int i = 0; i < retry; i++) {
    try {①
        LOG.info("Calling rules service {}", rulesUrl);
        items = restTemplate.getForObject(rulesUrl,
                                         ItemDto[].class, id, cartIds);
        itemIds = itemsToCommaString(items);
        LOG.info("Inventory items {}", itemIds);
        break;②
    } catch (Exception e) {
        if (i == retry - 1) {③
            throw e;
        }
    }
}
```

- ① Retry loop
- ② Break out loop if succeeded
- ③ Rethrow exception if end of loop

In the java code we can implement a retry pattern using a for loop ① where we keep looping until either the operation succeeds ② or kept on failing till the end of the loop ③ . As you can see implementing this code is a bit cumbersome, error prone, and *ugly*. Isn't there a better way? Well if you are using Apache Camel then we have already shown you the elegance of Camel's error handler, but some of our microservices is not using Camel, so what we can do? We can use a pattern called Circuit Breaker which has become very popular with microservice architectures.

7.4.3 Using Circuit Breaker to handle failures

In the previous section we covered the retry pattern and how it can overcome transient errors, such as slow networks, or a temporary overloaded service, by retrying the same operation. But sometimes failures are not transient, such as a downstream service may not be available, be overloaded, or has a bug in the code causing application level exceptions. In those situations there is no benefit in retrying the same operation and putting additional load on an already struggling system. If we don't deal with these situations, we risk degrading our own service, holding up threads, locks, 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 this in a graceful manner until the service is back and healthy again. What we 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 which was described by Michael Nygard in his *Release It!* book.

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 electric equipment.

The software based circuit breaker works on the same notion by encapsulating the operation and monitoring it for failures. The circuit breaker operates in three states as illustrated in figure 7.11.

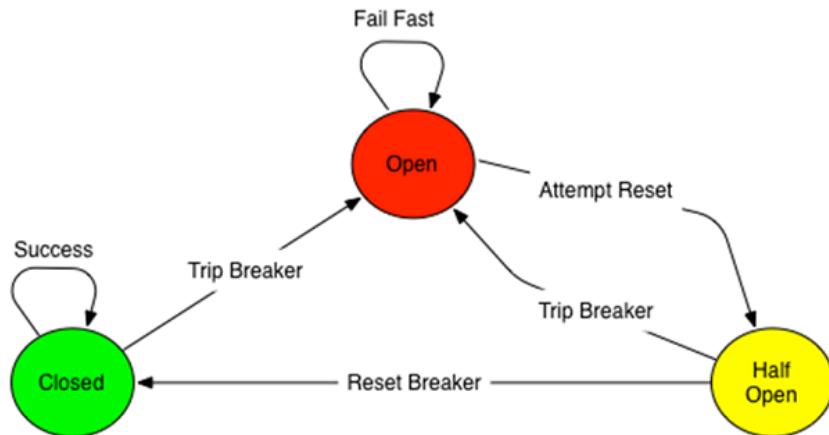


Figure 7.11 - Circuit Breaker pattern operates in three states. The closed state (green) when the operation operates successfully. Upon a number of successive failures the breaker trips into the open state (red) and fails fast. After a short period the breaker attempts to reset in the half open state (yellow), if its successful then the break reset into closed state (green), and in case of failure transfers back to open state (red).

The states are as follows:

- *Closed* - When operating successfully.
- *Open* - When failure 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 attempt to see if it can complete successfully, and depending on the outcome transfer to either open or closed state.

Speaking from practical experience with using the circuit breaker you may mixup 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 as both makes 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 as opposed to circuit breaker preventing overloading struggling downstream systems. The former is good for dealing with transient failures and the latter for more permanent and long-lasting failures.

In a 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. However one does not exclude the other both patterns can be used to deal with failures.

So 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 was an extension to the load balancer pattern. This implementation comes out of the box from camel-core and provides a basic implementation. However in recent time the Netflix Hystrix library has become popular and therefore the Camel team integrated Hystrix as a native EIP pattern in Camel. We recommend using the latter which is also what is being 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. Another way of describing Hystrix is by the following items:

- Provides protection against services which are unavailable
- Provides timeout to guard against unresponsive services
- Separation of resources using bulk head pattern
- Shedding loads and failing fast instead of queueing
- Self healing
- Real time monitoring

We will in the following touch each of the bullet items, but lets start with a simple Hystrix example.

USING HYSTRIX WITH PLAIN JAVA

Hystrix provides the `HystrixCommand` class which you extend and implement the `run` method containing your potential faulty code, such as a remote network call. Listing 7.17 shows how this is done.

Listing 7.17 - Creating a custom Hystrix Command

```
public class MyCommand extends HystrixCommand<String> {  
    public MyCommand() {  
        super(HystrixCommandGroupKey.Factory.asKey("MyGroup"));  
    }  
  
    protected String run() throws Exception {  
        COUNTER++;  
        if (COUNTER % 5 == 0) {  
            throw new IOException("Forced error");  
        }  
        return "Count " + COUNTER;  
    }  
}
```

- ① Extending `HystrixCommand`
- ② Specify group
- ③ All unsafe code goes into `run` method
- ④ Simulated error every 5th call

As you can see we have created our custom Hystrix command in the `MyCommand` class extending `HystrixCommand<String>` ① with `String` specified as 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 potential unsafe and fault code is put inside the `run` method ③ . In this example we have built in the code a failure that happens for every 5th call to simulate some kind of error ④ .

CALLING A HYSTRIX COMMAND FROM JAVA

So how do you use this command? Well that is very simple with Hystrix, you just create an instance of it, and call the ... no not the `run` method, but the `execute` method as shown:

```
MyCommand myCommand = new MyCommand();  
String out = myCommand.execute();
```

.. and the result of running this command would be

```
Count 1
```

If you run it again like:

```
MyCommand myCommand = new MyCommand();  
String out = myCommand.execute();  
String out2 = myCommand.execute();
```

... then you would expect:

```
Count 1
Count 2
```

... but that is not what happens, instead you have:

```
Count 1
com.netflix.hystrix.exception.HystrixRuntimeException: MyCommand command executed multiple
times - this is not permitted
```

That is an important aspect with Hystrix that each command is not reusable, you can only use a command once and only once. That also means you cannot store any state in a command as it can only be called once, and there is no state leftover for successive calls.

You can find this example from the source code in chapter7/hystrix directory which you can try using the following Maven goal:

```
mvn test -Dtest=MyCommandTest
```

You may have noticed in listing 7.17 the COUNTER instance, which is used as global state stored outside the Hystrix command, and the fact the example was constructed to fail every 5th call. How do you deal with exceptions thrown from the run method?

ADDING Fallback

When the run method cannot be executed successfully then we can use Hystrix built-in fallback method to return an alternative response. For example to add a fallback to listing 7.17 you would simply override the getFallback method as follows:

```
protected String getFallback() {
    return "No Counter";
}
```

Running this command ten times would yield 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 was constructed to fail every 5th time which is when the fallback kicks in and returns the alternative response.

You can find this example from the source code in chapter7/hystrix directory which you can try using the following Maven goal:

```
mvn test -Dtest=MyCommandWithFallbackTest
```

Okay so what about using Hystrix with Camel?

7.4.5 Using Hystrix with Camel

Camel makes it easy to use Hystrix. All you basically have to do is to add the camel-hystrix dependency to your project, and then use the Hystrix EIP in your Camel routes.

Lets migrate the previous example from section 7.4.4 to use Camel. The code from listing 7.17 which was implemented in the run method of `HystrixCommand` is simply refactored into a plain Java bean as shown:

```
public class CounterService {
    private int counter; 1

    public String count() throws IOException { 2
        counter++;
        if (counter % 5 == 0) {
            throw new IOException("Forced error");
        }
        return "Count " + counter;
    }
}
```

- 1 Storing counter state in the bean
- 2 The unsafe code

The code which was previously in the run method of the `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 is not a `HystrixCommand` implementation. From section 7.4.4 we learned that its a requirement for a Hystrix command to be stateless and only executable once, which forced us to store any state outside the `HystrixCommand`.

Using Hystrix in Camel routes is so easy as its just a few line of codes as shown below:

```
public void configure() throws Exception {
    from("direct:start")
        .hystrix() 1
        .to("bean:counter")
        .end() 2
        .log("After calling counter service: ${body}");
}
```

- 1 Hystrix EIP
- 2 End of Hystrix block

In the route we use Hystrix to denote the beginning of the protection of Hystrix. Any of the following nodes are run from within a `HystrixCommand`. In this example that would be calling the counter bean. To denote when the Hystrix command ends we 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 2nd bean you can do:

```
.hystrix()
    .to("bean:counter")
    .to("bean:anotherBean")
.end()
```

Using Hystrix in XML is just as easy. The equivalent route is shown in listing 7.18.

Listing 7.18 - Using Hystrix EIP using XML DSL

```
<bean id="counterService" class="camelinaction.CounterService"/> ①

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <hystrix>
            <to uri="bean:counterService"/> ②
        </hystrix>
        <log message="After calling counter service: ${body}"/> ③
    </route>
</camelContext>
```

- ① Define counter bean
- ② Hystrix EIP
- ③ End of Hystrix block

In XML DSL we first define the counter service bean as a Spring `<bean>` ① . In the Camel route we can easily use Hystrix using `<hystrix>` ② . The nodes inside the Hystrix block ③ are then run under the control from a `HystrixCommand`.

So what about using fallbacks?

ADDING FALBACK

Using Hystrix fallback with Camel is also very easy to do as shown:

```
public void configure() throws Exception {
    from("direct:start")
        .hystrix()
            .to("bean:counter")
            .onFallback()
                .transform(constant("No Counter"))
            .end()
        .end()
    .log("After calling counter service: ${body}");
}
```

- ① Hystrix fallback

Fallback is added using `.onFallback()` where the following nodes are included. In this example we just perform a message transformation to set the message body to the value of "No Counter".

Likewise using fallback with XML DSL is very easy as well:

```
<route>
    <from uri="direct:start"/>
```

```

<hystrix>
    <to uri="bean:counterService"/>
    <onFallback>
        <transform>
            <constant>No Counter</constant>
        </transform>
    </onFallback>
</hystrix>
<log message="After calling counter service: ${body}"/>
</route>

```

①

① Hystrix fallback

You can find this example with the source code in the chapter7/camel-hystrix example which you can try using the following Maven goals:

```

mvn test -Dtest=CamelHystrixWithFallbackTest
mvn test -Dtest=SpringCamelHystrixWithFallbackTest

```

One aspect which we haven't touched very much is the fact that each `HystrixCommand` must be uniquely defined using a command key. Hystrix enforces this in the constructor of the `HystrixCommand` as we did in listing 7.17 ② when we were not using Camel. But how to do this when Camel with Hystrix?

CONFIGURING COMMAND KEY

Each Hystrix EIP in Camel are uniquely identified using their 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 auto assigned node ids which typically is named using the node and a counter such as `hystrix1`, `bean1`, `bean2`.

This means the command key of the Hystrix EIP would be `hystrix1`. If you want to assign a specific key then you need to configure the node id with the value as shown in bold:

```

.hystrix().id("MyHystrixKey")
    .to("bean:counter")
    .to("bean:anotherBean")
.end()

```

In XML DSL you would do as follows:

```

<hystrix id="MyHystrixKey">
    <to uri="bean:counterService"/>
    <to uri="bean:anotherBean"/>
</hystrix>

```

Speaking of configuring Hystrix. Hystrix has a wealth of configuration options which you can use to tailor for all kind of behaviors. Some of these options are more important to know than others, which we will learn about bulk head pattern.

But first lets cover the basics of configuring Hystrix.

7.4.6 Configuring Hystrix

Configuring Hystrix is not a simple task. There is a lot of different options that can tweak all kind of details how the circuit breaker should operate. All these options all play a purpose to allow very fine grained configuration. Your first encounters with all these options will leave you baffled and it takes time to learn the nuance between the options and which role they play. As a rule of thumb the out of the box settings are a good compromise and there is a few options that you most often would be using.

In this section we will show you how to configure Hystrix in pure Java and as well with Camel and Hystrix.

CONFIGURING HYSTRIX WITH JUST JAVA

You configure a `HystrixCommand` in the constructor using a *somewhat special* fluent builder style.

For example to configure a command to open the circuit breaker if we get ten or more failed requests within a rolling time period of 5 seconds, we can configure this as follows:

```
public MyConfiguredCommand() {
    super(Setter
        .withGroupKey(HystrixCommandGroupKey.Factory.asKey("MyGroup"))
        .andCommandPropertiesDefaults(
            HystrixCommandProperties.Setter()
                .withCircuitBreakerRequestVolumeThreshold(10)
                .withMetricsRollingStatisticalWindowInMilliseconds(5000)
        ));
}
```

Yes it takes some time to learn how to configure Hystrix using this configuration style. There is plenty more options than shown here, and we refer you to the Hystrix documentation (<https://github.com/Netflix/Hystrix/wiki>) to learn more about each and 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()
        .circuitBreakerRequestVolumeThreshold(10) ①
        .metricsRollingPercentileWindowInMilliseconds(5000)
    .end() ②
    .to("bean:counter")
```

```
.end()
```

③

- ① Hystrix configuration
- ② End of Hystrix configuration
- ③ 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 lets take a moment to highlight the options we think are of importance.

IMPORTANT HYSTRIX OPTIONS

Table 7.2 lists the Hystrix options we think is worthwhile to learn to use first.

Table 7.2 - Important Hystrix options

| Option | Default Value | Description |
|---|---------------|--|
| commandKey | node id | Used to identify the Hystrix command. This option cannot be configured but is locked down to be the node id to make the command unique. |
| groupKey | CamelHystrix | Used to identify the Hystrix group being used by the EIP to correlate statistics, circuit-breaker, properties, etc. |
| circuitBreakerRequestVolumeThreshold | 20 | This property sets the minimum number of requests in a rolling window that will trip the circuit. If below this number the circuit will not trip regardless of error percentage. |
| circuitBreakerErrorThresholdPercentage | 50 | Error percentage threshold (as whole number such as 50) at which point the circuit breaker will trip open and reject requests. It will stay tripped for the duration defined in the <code>circuitBreakerSleepWindowInMilliseconds</code> option. |
| circuitBreakerSleepWindowInMilliseconds | 5000 | The time in milliseconds after a circuit breaker trips open that it should wait before trying requests again. |
| metricsRollingStatisticalWindowInMilliseconds | 10000 | Duration of statistical rolling window in milliseconds. This is how long metrics are kept for the thread pool. |

| | | |
|--------------------------------|------|---|
| corePoolSize | 10 | This property sets the core thread-pool size. This is the maximum number of Hystrix commands that can execute concurrently. |
| maxQueueSize | -1 | This property sets the maximum queue size of the thread pool task queue. |
| executionTimeoutInMilliseconds | 1000 | Time in milliseconds at which point the command will timeout and halt execution. |

Some of the the options from table 7.2 will be covered with the next topic about the Bulkhead pattern.

7.4.7 Bulkhead pattern

Imagine a situation where a downstream service is under pressure and your application keeps calling the downstream service. The service is becoming latent but not enough to trigger either a timeout in your application nor to 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. In such situations we want to limit the latency to only the dependency that is 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 basically a separation of resources such that one set of resources does not impact others. This pattern is well known in ships to create watertight compartments that can contain water in the case of a hull breach. A famous tragedy is Titanic which bulk heads were to low and didn't prevent water from leaking into adjacent compartments and 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, then the thread pool can become fully utilized and fail fast by rejecting new requests to the command. The thread pool is isolated from each other and therefore other commands can still operate successfully.

The bulkhead is enabled by default and each command is assigned a thread pool of 10 worker threads by default. The thread pool has no task pool as backlog and therefore if all 10 threads is utilized, Hystrix will reject processing new requests and therefore fail fast.

The option corePoolSize can be used to configure number of threads in the pool, and task queue is enabled by setting the maxQueueSize option to a positive size.

Another powerful feature from Hystrix is the execution timeout.

TIMEOUT WITH HYSTRIX

Every Hystrix command is executed within the scrutiny of a timeout period. If the command does not complete within the given time, Hystrix will react and cause the command to fail. This timeout is 100% controller by Hystrix and should not be mistaken with any timeouts from Camel components, or downstream services. In other words Hystrix comes with timeout out of

the box. The default value is very sensitive at only 1 second. One second is a low timeout and therefore you may find yourself often configure this value to a value that suits your use-cases.

The timeout is listed last in table 7.2 and can be configured as shown in Java DSL:

```
from("direct:start")
    .hystrix()
        .hystrixConfiguration().executionTimeoutInMilliseconds(2000).end() ①
            .toD("direct:${body}")
        .end();
```

① Two seconds as timeout

... and in XML DSL:

```
<route>
    <from uri="direct:start"/>
    <hystrix>
        <hystrixConfiguration executionTimeoutInMilliseconds="2000"/> ①
            <toD uri="direct:${body}"/>
        </hystrix>
    </route>
```

① Two seconds as timeout

The source code for this book includes an example using Hystrix to call downstream Camel routes which processes either one or three seconds before responding. This demonstrates Hystrix with a timeout value of two seconds will succeed using a fast response and fail with a timeout exception when using a slow response. The example is located in the chapter7/camel-hystrix directory which you can try using the following Maven goals:

```
mvn test -Dtest=CamelHystrixTimeoutTest
mvn test -Dtest=SpringCamelHystrixTimeoutTest
```

When a Hystrix timeout occurs then the Hystrix command is regarded as failed and the Camel route fails with a timeout exception.

TIMEOUT AND FAILBACK WITH HYSTRIX

If a timeout error happens you may want to let Hystrix handle this by a fallback to setup an alternative response.

This can easily be done with Camel using `onFallback` as shown below:

```
from("direct:start")
    .hystrix()
        .hystrixConfiguration()
            .executionTimeoutInMilliseconds(2000)
        .end()
        .log("Hystrix processing start: ${threadName}")
        .toD("direct:${body}")
        .log("Hystrix processing end: ${threadName}")
    .onFallback()
        .log("Hystrix fallback start: ${threadName}") ②
```

```
.transform().constant("Fallback response")
.log("Hystrix fallback end: ${threadName}")
.end()
.log("After Hystrix ${body}");
```



① Hystrix fallback when any error happens

You can find this example with the source code in chapter7/hystrix-camel directory which can be run using the following Maven goals:

```
mvn test -Dtest=CamelHystrixTimeoutAndFallbackTest
mvn test -Dtest=SpringCamelHystrixTimeoutAndFallbackTest
```

If you run the example then the fast and slow response tests will output the following to the console.

The fast response logs:

```
2017-01-01 20:21:14,654 [-CamelHystrix-1] INFO route1
  - Hystrix processing start: hystrix-CamelHystrix-1
2017-01-01 20:21:14,660 [-CamelHystrix-1] INFO route2
  - Fast processing start: hystrix-CamelHystrix-1
2017-01-01 20:21:15,662 [-CamelHystrix-1] INFO route2
  - Fast processing end: hystrix-CamelHystrix-1
2017-01-01 20:21:15,663 [-CamelHystrix-1] INFO route1
  - Hystrix processing end: hystrix-CamelHystrix-1
2017-01-01 20:21:15,666 [main] INFO route1
  - After Hystrix Fast response
```

And the slow response logs:

```
2017-01-01 20:23:08,035 [-CamelHystrix-1] INFO route3
  - Slow processing start: hystrix-CamelHystrix-1
2017-01-01 20:23:10,023 [HystrixTimer-1] INFO route1
  - Hystrix fallback start: HystrixTimer-1
2017-01-01 20:23:10,023 [HystrixTimer-1] INFO route1
  - Hystrix fallback end: HystrixTimer-1
2017-01-01 20:23:10,026 [main] INFO route1
  - After Hystrix Fallback response
```

So 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 is processing. In the test with the fast response its the same thread `CamelHystrix-1` which routes the Camel message in Hystrix command (Hystrix EIP). The last log lines is outside the Hystrix EIP and therefore outside the Hystrix command and therefore its the caller thread that is 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 Hystrix command is separated and being processed by separated threads.

Figure 7.12 illustrates this principle.

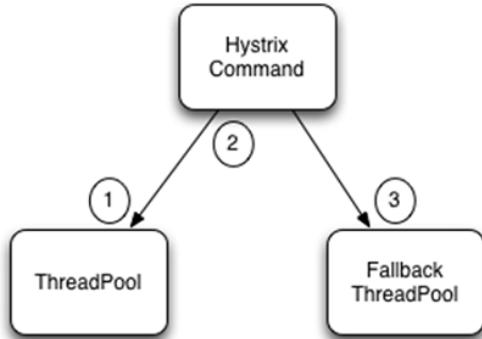


Figure 7.12. The hystrix command uses a thread from the regular thread pool (1) to route the Camel message. The task runs and upon completion the command completes successfully (2). However the task can also fail due to a timeout or an exception was thrown during Camel routing which causes the hystrix 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.

There is one important caveat with Hystrix fallbacks. In the example you may have wondered why the fallback thread name was named `HystrixTimer-1`. That is because it was Hystrix that triggered a timeout and executed the fallback. This timeout thread is from the timeout functionality within Hystrix that is shared across all the Hystrix commands. Now we are down to what the caveat is about. Your Hystrix fallbacks should be written as short methods that does 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. We will cover this in chapter 17 when we return back to microservices in a distribution environment.

FALLBACK VIA NETWORK

If the fallback must do a network call the you must use another Hystrix command which comes with its own thread pool to ensure total thread isolation. With Camel this is easy because we have implemented a special fallback for network calls which is named `onFallbackViaNetwork` as shown below:

```

from("jetty:http://0.0.0.0/myservice")
    .hystrix()
        .to("http://server-one")
        .onFallbackViaNetwork()
            .to("http://server-backup")
  
```

①

① Fallback with a network call

The route above exposes a HTTP service using Jetty which proxies to another HTTP server on `http://server-one`. If the downstream service is not available then Hystrix will fallback to another downstream service over the network at `http://server-backup`.

Fallback is not an one-size-fits-all solution

A Hystrix fallback is not a silver bullet that solves every problem in a microservices or distributed architecture. It is very domain specific and case by case whether a fallback is applicable or not. In the Rider Auto Parts example with the recommendation service a fallback can be very 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 too slow to respond? We can degrade to a generic list for users in a particular region; or just a generic list.

However in other domains such as a flight booking system a service that performs the actual booking at the airline could likely not fallback because a flight seat cannot be guaranteed to have been booked with the airline.

With fresh and new knowledge in your tool belt you arrange for another weekend where the wife and kids are away on a retreat. To have a good time you have stocked with a fresh six pack of beers and a good scotch just in case you run out of the beers.

7.4.8 Calling other microservice with fault tolerance

The previous prototype you build was a set of microservices which collectively implemented a recommendation system for Rider Auto Parts. However the prototype was not designed for failures. So how can you apply fault tolerance to those microservices? What follows transpires what you did during the weekend to add fault tolerance to the your prototype.

Figure 7.13 illustrates the collaborations among the microservices and where we are in trouble.

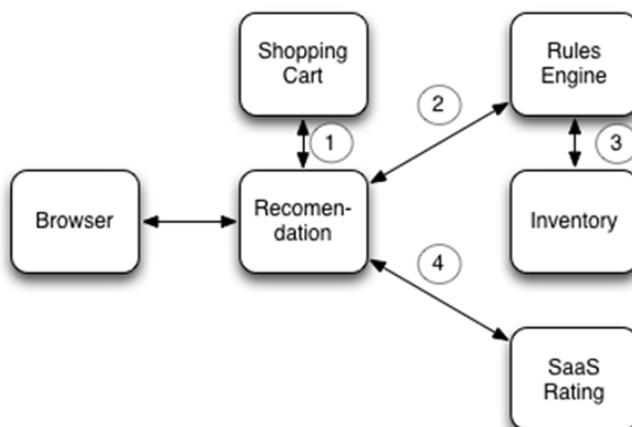


Figure 7.13 - Rider Auto Parts recommendation system. Each arrow represents a dependency among the microservices. Each number represent where we need to add fault tolerance.

From figure 7.13 you can see that you need to add fault tolerance four times (1 2 3 4) to the prototype.

In the following we will cover what you did to implement fault tolerance at those four places in the prototype.

We start from the top with the recommendation microservice.

USING HYSTRIX WITH SPRING BOOT

The recommendation microservice is a Spring Boot application that does not 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>1.2.3.RELEASE</version>
</dependency>
```

The support for Hystrix comes with the Spring Cloud project and hence the name of the dependency.

Support for Hystrix on Spring Boot requires you to implement a `@Service` class which holds the code to run as a Hystrix command. Listing 7.18 shows what you did to add fault tolerance at number 1 from figure 7.13.

Listing 7.13 - Using Hystrix in Spring Boot as a @Service class

```
@Service
public class ShoppingCartService { 1

    private final RestTemplate restTemplate = new RestTemplate();

    @HystrixCommand(fallbackMethod = "emptyCart")
    public String shoppingCart(String cartUrl, String id) { 2
        CartDto[] carts = restTemplate.getForObject(cartUrl,
            CartDto[].class, id); 3
        String cartIds = Arrays.stream(carts)
            .map(CartDto::toString)
            .collect(Collectors.joining(","));
        return cartIds; 4
    }

    public String emptyCart(String cartUrl, String id) { 5
        return "";
    }
}
```

- 1 Annotate class as a Spring `@service` class
- 2 Mark method as Hystrix command
- 3 Call downstream service
- 4 Convert response to a comma separated String
- 5 Method used as Hystrix fallback

In listing 7.13 you created a new class `ShoppingCartService` which is responsible for calling the downstream shopping cart service. The class has been annotated with `@Service` (or `@Component`) which is required by Spring when using Hystrix. The `@HystrixCommand` annotation ❷ is added on the method which will be wrapped as a Hystrix command. This means that all the code that runs inside this method will be under the control of Hystrix. Inside this method we call the downstream service ❸ using Spring's `RestTemplate` to perform a REST call. Having a response we transform this into a comma separated string using Java 8 *stream magic* ❹. On the `@HystrixCommand` annotation we have specified the name of the fallback method which 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 followed same practice to implement fault tolerance to calling the rules and rating microservices (number ❷ and ❹ from figure 7.13).

The last code change needed is in the `RecommendController` class which orchestrates the recommendation service. Listing 7.18 lists the updated source code.

Listing 7.18 - Recommend controller using the fault tolerant services

```

@EnableCircuitBreaker                               ❶
@RestController
@RequestMapping("/api")
@ConfigurationProperties(prefix = "recommend")
public class RecommendController {

    private String cartUrl;
    private String rulesUrl;
    private String ratingsUrl;

    @Autowired
    private ShoppingCartService shoppingCart;          ❷
    @Autowired
    private RulesService rules;                      ❷
    @Autowired
    private RatingService rating;                   ❷

    @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);      ❸
        ItemDto[] items = rules.rules(rulesUrl, id, cartIds);           ❹
        String itemIds = itemsToCommaString(items);

        RatingDto[] ratings = rating.rating(ratingsUrl, itemIds);      ❺
        for (RatingDto rating : ratings) {
            appendRatingToItem(rating, items);
        }
    }
}

```

```

        return Arrays.asList(items);
    }
}

```

The annotation `@EnableCircuitBreaker` ① is used to turn on Hystrix with Spring Boot. You have then dependency injected the three classes with the Hystrix command ② which will be used from the controller to call the downstream microservices. Because the logic to call those downstream services are now moved into those separate classes the code in the recommend method is simpler. All you do is to call those services in order ③ ④ ⑤ as if it was regular Java method calls. However under the covers those method calls will be under the wings of Hystrix wrapped in as a Hystrix command with fault tolerance included in the batteries.

TRYING THE EXAMPLE

Now you are ready to try this in action. So what you did was to run the code from the chapter7/prototype2/recommend directory using the following goal from Maven:

```
mvn spring-boot:run
```

And then open a web browser on `http://localhost:8080/api/recommend` which returned 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 are not available then the application returns a fallback response. The fallback response is created because of these conditions:

- The `ShoppingCartService` - Returns an empty cart as response
- The `RulesService` - Returns the special item with no 999 as fallback
- The `RatingService` - Gives each item a rating of 6 as fallback

Now you turn on each of these downstream service 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
```

and then 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 as before which is not 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 recommendation service the response is now:

```
[{"itemNo":999,"name":"special","description":"Special Rider Auto Part premium service","number":100,"rating":6}]
```

Oh boy its still the same response, why is that? So you do what every experienced developer always does when something is not right. You go home ... but you are already home, so what can you do? You take a little break and grabs a beer from the six-pack. Dear reader you are almost at the end of this chapter, so you don't have to take a break just yet, but you are 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:

```
2017-01-06 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 is not fault tolerant and fails because it cannot connect to the inventory service over JMS. This is the connection at number 3 from figure 7.13. Before you start adding hystrix to the rules service lets 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
```

and by refreshing the call to the rating service the response is now:

```
[{"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 run its:

```
[{"itemNo":999,"name":"special","description":"Special Rider Auto Part premium service","number":100,"rating":98}]
```

There is a subtle change in the rating. The rating service was implemented to return a random number. Notice the first rating was 24 and the second is 98.

Okay the last piece of the puzzle is to add fault tolerance to the rules service which is implemented using WildFly Swarm with Camel. So lets 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 7.13. This service is using Camel running on WildFly Swarm. To use Hystrix you have to add the camel-other fraction as a Maven dependency in the pom.xml file:

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>camel-other</artifactId>
</dependency>
```

Using Hystrix with Camel is easy as you already have learned, so all you have done is to modify the Camel route as shown below:

```
public void configure() throws Exception {
    JaxbDataFormat jaxb = new JaxbDataFormat();
    jaxb.setContextPath("camelinaction");

    from("direct:inventory")
        .hystrix()
        .to("jms:queue:inventory")
        .unmarshal(jaxb)
        .onFallback()
        .transform().constant("resource:classpath:fallback-inventory.xml");
}
```

- ① Setup JAXB for XML
- ② Hystrix EIP
- ③ Hystrix fallback

Because the response from the inventory service is in XML format, and this route is expected to return data as POJO you need to setup JAXB ① which is later used to unmarshal from XML to POJO. The call to the inventory service is protected by Hystrix circuit breaker ②. In case of any failures then the fallback is executed ③ which loads a static response from a 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 it was just adding a maven dependency, and then three lines of code in the Camel route, and create a XML file with the fallback response.

You have now implemented fault tolerance using Hystrix EIP in all the microservices at the numbers ① ② ③ and ④ from figure 7.13. So lets try the complex example.

CONTINUE TRYING THE EXAMPLE

To continue trying the prototype you start the rules microservice again, now that it has been updated with fault tolerance. From the shell you type:

```
cd chapter7/prototype2/rules2
mvn wildfly-swarm:run
```

and 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":32}]
```

The response looks familiar. Its the fallback response from the rules service. But didn't we add fault tolerance to this? So why is it doing this? Just as you grabbed another beer from the six-pack you realize that the rules service calls the downstream inventory service, which is represented as number ❸ in figure 7.13. And its failing with a fallback response because you have not yet started the inventory service. So you quickly open another shell and type:

```
cd chapter7/prototype2/inventory2
mvn compile exec:java
```

And you just happened to glace at the logs from the rules service and noticed that it stopped logging stacktraces and logged that it had successfully re-connected to the message broker, which was just started as part of the inventory service.

```
2017-01-06 12:01:56,558 INFO
[org.apache.camel.component.jms.DefaultJmsMessageListenerContainer]
(Camel (camel-1) thread #1 -
TemporaryQueueReplyManager[inventory]) Successfully refreshed JMS Connection
```

So with that in mind you are confident that 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 if the rules service is fault tolerant when the inventory service is not available. So 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 is the fallback response from the rules service which would be intended to return a list of known items that Rider Auto Parts always have on stock.

You still have one beer left from the six-pack and the Whisky to celebrate your success. So lets 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 very easy to use Camel and Hystrix on Spring Boot, even after having five out of six beers from the six-pack. So what we want to do is to migrate the rules microservices from using WildFly Swarm with Camel and Hystrix to use Spring Boot with Camel and Hystrix.

The first thing you do is to setup 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 which leverages Spring @RestController ① instead of JAX-RS used by WildFly Swarm as shown in listing 7.19.

Listing 7.19 - Spring REST controller calling a Camel route using direct endpoint

```
@RestController
@RequestMapping("/api")
public class RulesController {

    @Produce(uri = "direct:inventory")
    private FluentProducerTemplate producer; ①

    @RequestMapping(value = "rules/{cartIds}",
                    method = RequestMethod.GET, produces = "application/json")
    public List<ItemDto> rules(@PathVariable String cartIds) {
        ItemsDto inventory = producer.request(ItemsDto.class); ②

        return inventory.getItems().stream() ③
            .filter((i) -> cartIds == null
                     || !cartIds.contains("") + i.getItemNo()))
            .sorted(new ItemSorter())
            .collect(Collectors.toList());
    }
}
```

- ① Spring REST controller
- ② Inject Camel ProducerTemplate
- ③ Call Camel route
- ④ Filter, sort and build response as list

The controller class uses Camel FluentProducerTemplate ② to call the Camel route ③ which does the actual call of the downstream service using Hystrix. The response from the Camel route is then filtered, sorted and collected as a List ④ using the Java 8 stream API.

The Camel route that leverages Hystrix EIP is show in listing 7.20.

Listing 7.20 - Camel route using Hystrix to call downstream service using JMS

```
@Component
public class InventoryRoute extends RouteBuilder {

    public void configure() throws Exception {
        JaxbDataFormat jaxb = new JaxbDataFormat();
        jaxb.setContextPath("camelinaction"); ①
    }
}
```

```

from("direct:inventory")
    .hystrix() ②
        .to("jms:queue:inventory")
        .unmarshal(jaxb)
        .onFallback()
            .transform().constant("resource:classpath:fallback-inventory.xml");
    }
}

```

- ① Setup JAXB for XML
- ② Hystrix EIP
- ③ Hystrix fallback

Migrating the Camel route shown in listing 7.20 from WildFly Swarm to Spring Boot only required one change. The class annotation is using Spring's `@Component` instead of WildFly Swarm using CDI's `@ApplicationScoped`. That's all what was needed. That is a testimony to Camel is indifferent and works on any container as-is.

The last migration effort to do is to setup Spring Boot to use ActiveMQ and setup the connection to the message broker. You did this by adding the following dependency to Maven `pom.xml` file:

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

Then you setup the connection to the ActiveMQ broker in the `application.properties` file as shown:

```
spring.activemq.broker-url=tcp://localhost:61616
server.port=8181
```

The other setup you did was 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 are almost fooled how the Camel route in listing 7.20 knows about the Spring Boot ActiveMQ broker setting we just did in the `application.properties` file. Well that is because the Camel JMS component is using Spring JMS under the hood. And Spring JMS has not surprisingly support for Spring Boot and can automatic wire up to the ActiveMQ broker.

You are now ready to run this example. All the other microservices has been stopped so when you start this example its 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}]
```

This is expected because the inventory service is not running and therefore there is no ActiveMQ broker running 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 calls 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 are almost done for the day but just wanted to test the fault tolerance so you stop the inventory service which 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 is still time for one more adventure before the weekend comes to a close. The prototype includes four communication points which are protected by Hystrix circuit breakers (figure 7.13). When the recommendation service is called then there are potential four service calls that can fail and return a fallback response. So how do you know where there are problems? That is a good question and its whole another topic about monitoring and distributed systems which we will cover much more in chapter 16 and 17.

After six beers we don't want to leave you hanging so lets embark on a 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 feed into the dashboard.

The weekend is coming to a close and you just want to quickly try to see what it would take to setup 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*. This is 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>1.2.3.RELEASE</version>
</dependency>
```

The Spring Boot Actuator enables real time monitoring capabilities such as health checks, metrics and also support for Hystrix streams. The actual 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 can be done by adding the `@EnableCircuitBreaker` annotation to your SpringBoot application class.

```
@EnableCircuitBreaker
@SpringBootApplication
public class SpringbootApplication {
```

The Hystrix EIP from the Camel route is automatic 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 are now ready to run the microservice. First you run the inventory service which the rules microservice communicate with.

```
cd chapter7/prototype2/inventory2
mvn compile exec:java
```

Then 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 are ready to have fun and install the Hystrix dashboard.

INSTALLING THE HYSTRIX DASHBOARD

The dashboard can more easily be installed by downloading the standalone JAR from the following github url: <https://github.com/kennedyoliveira/standalone-hystrix-dashboard>.

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 you open the url: `http://localhost:7979/hystrix-dashboard`. On the dashboard you type in: `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 is graphics.

TIP WildFly Swarm have also made it easy to install the Hystrix Dashboard as a .war on WildFly. There is little more information at: <http://wildfly-swarm.io/tutorial/hystrix/>

Because there are no activity the graph is not very exciting, its all just zeros and a flat line. So lets turn up the load, and to do that you hacked up a little bash script that calls the rules service repetitively 10 times per second. You run the script by:

```
cd chapter7/prototype2/rules2-springboot-hystrixdashboard
chmod +x hitme.sh
./hitme.sh
```

and now there is activity in the dashboard.

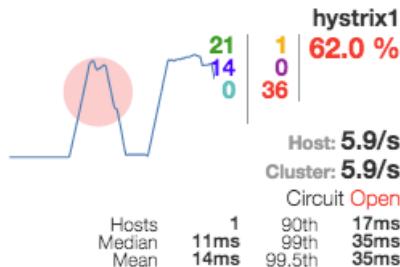
NOTE Understanding what all the numbers on the Hystrix dashboard represent takes time to learn. You can find information on the internet that explains the dashboard in much more detail.

After the script has been running for a while its all green and happy. So you go out on a limp 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 7.14.

Hystrix Stream: <http://localhost:8181/hystrix.stream>

Circuit

Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)



Thread Pools

Sort: [Alphabetical](#) | [Volume](#) |

CamelHystrix

| | Active | Max Active | 1 |
|-----------|--------|------------|----|
| | Queued | Executions | 62 |
| Pool Size | 10 | Queue Size | 5 |

Figure 7.14 - Hystrix Dashboard visualizing the circuit breaker running in the rules microservice. The state of the breaker is open due to recent number of errors because the downstream inventory service has been stopped.

After a little while you start the inventory service again and see that the changes in the dashboard. The red numbers should 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. Its late in the evening and you have done well. So you pour yourself a neat glass of whiskey and sit and reflect what you have archived this time.

WHAT HAVE YOU LEARNED THIS TIME

You learned that in an microservices and distributed architecture its even more important to design for errors. With so many individual services that communicates across the network in a mesh its destined for errors to happen. Luckily Camel makes it easier to incorporate error handling strategies using Camel's error handler or Hystrix circuit breakers.

Despite circuit breakers are easy to use with Camel or the `@HystrixCommand` then they are not a silver bullet solution for everything. The mantra about use the right tool for the right job applies. However circuit breakers are 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 timeouts that matches your use-cases. Hystrix comes with a very 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 in your individual microservices its 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 different 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 is a big and popular topic these days in 2017, which we will cover in chapter 17. At first you need to master many other aspects about Camel.

The weekend is over and so is this chapter. Thanks for sticking with us all the way.

7.5 Summary and best practices

That was a long and bumpy ride to get to the end of this chapter. We have enjoyed the journey. Writing this chapter 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 much caught up in the microservices buzzword bingo. Microservices is a great concept but its not a universal super solution to every software problem. Don't panic because a number of internet startups have been successful with microservices and are able to do all the jazz and deploy to production hundred times per day. For regular enterprises we are likely going to see a middle ground with a enterprise microservice that works the best for them.

What we have been focusing on in this book and this chapter is to cover how you can do microservice development with Camel. In the start of the chapter in section 7.1 we outlined eight microservice characteristics which we believe are most applicable for Camel developers. We encourage you to take a second look on these eight after you have reached the end of this

chapter. In fact add a bookmark and circle back to this section from time to time. We want you to have those concepts in your tool belt

As usual we have put together a bulleted list with best practices and advices.

- *Microservices is more than technology* - You cannot just build smaller applications and use REST as communication protocol and think you are doing microservices. There is a lot more to master on both technical and non-technical levels. The biggest challenges in organizations that want to move to doing microservices is potentially with frictions from their culture, organizational structure, team communication, and old habits.
- *Small in size* - A good rule of thumb is the two-pizza-rule. A microservice should be no bigger than a team of people can get fed by two pizzas and be self sustaining and autonomous being able to manage all aspects of the service.
- *Design for failures* - Microservices are inherently a distributed architecture. Every time you have remote network communication failures can happen. Design your microservices as fault tolerant. Camel offers great capabilities in this matter with its error handler and first class support for Hystrix circuit breakers.
- *Self monitoring* - With an increasing number of running microservices it becomes very 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 will cover 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 should be environmental specific configurable.
- *Testable* - Microservices should be quick to test and as best testing should be automated as part of a CI/CD pipeline. Chapter 9 covers testing and chapter 17 will showcase a microservices platform with CI/CD solution included.

The journey of microservices will be continued in chapter 17 where we talk about microservices in the cloud.

The next chapter takes you through how to develop Camel projects. You may think you have the skills already to build Camel projects and you are surely correct. However there is more to Camel than at first sight. Without giving away too much details the chapter helps new users whom are less familiar with Maven and Eclipse to ensure anyone can setup a development environment for build Camel projects. You also learn how to your own Camel components, data formats and more. And last but not least you learn a very important topic - how to debug your Camel routes.

8

Developing Camel projects

This chapter covers

- Creating Camel projects with Maven
- Creating Camel projects in the Eclipse IDE
- Debugging with Camel
- Creating custom components
- Creating custom interceptors
- Creating custom data formats
- Using the API component framework

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?

In this chapter, we'll show you how to build your own Camel applications. We'll go over the Maven archetype tooling that'll allow 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.

After that, we'll show you how to extend Camel by creating custom components, custom interceptors and data formats. Finally, we'll wrap up by showing you how to use Camel's API component framework, which can generate a near fully functional component just from scanning an arbitrary Java API.

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. In this section, we'll show you Camel's Maven archetypes, which are preconfigured templates for creating various types of Camel projects. After that, we'll talk 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 on here.

8.1.1 Using Camel Maven archetypes

Creating Maven-based projects is a pretty simple task. You mainly have to worry about creating a POM file and the various standard directories that you'll be using in your project. But if you're creating many projects, this can get pretty repetitive because there's a lot of boilerplate setup 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 on the official Maven website: <http://maven.apache.org/guides/introduction/introduction-to-archetypes.html>. Sonatype also provides a chapter on archetypes in the freely available *Maven: The Complete Reference* book: <https://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.

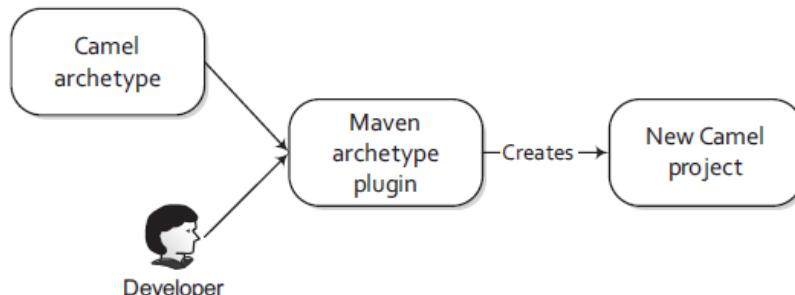


Figure 8.1 A Camel archetype and user input are processed by the Maven archetype plugin, which then creates a new Camel project.

To demonstrate how this works, let's look at the Maven quickstart archetype, which will generate a plain Java application (no Camel dependencies). It's the default option when you run this command:

```
mvn archetype:generate
```

The archetype plugin will ask you various questions, like 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
    |       '-- camelinaction
    |           '-- App.java
    '-- test
        '-- java
            '-- camelinaction
                '-- 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 convention over configuration paradigm, so locations are very important.

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 just 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 like `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 thirteen 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-activemq</code> | Creates a Camel project that has an embedded Apache ActiveMQ broker. |
| <code>camel-archetype-blueprint</code> | Creates a Camel project that uses OSGi blueprint. Ready to be deployed in OSGi. |
| <code>camel-archetype-component</code> | Creates a new Camel component. |

| | |
|--|--|
| <code>camel-archetype-api-component</code> | Creates a new Camel component which is based on some 3 rd party API. |
| <code>camel-archetype-cdi</code> | Creates a Camel project with Contexts and Dependency Injection spec (CDI) support. |
| <code>camel-archetype-dataformat</code> | Creates a new Camel data format. |
| <code>camel-archetype-groovy</code> | Creates a Camel project with a sample route in the Groovy DSL. |
| <code>camel-archetype-java</code> | Creates a Camel project that defines a sample route in the Java DSL. |
| <code>camel-archetype-scala</code> | Creates a Camel project with a sample route in the Scala DSL. |
| <code>camel-archetype-scr</code> | Creates a Camel project using the Apache Felix Service Component Runtime (SCR). Felix SCR is an implementation of the OSGi Declarative Services specification. Ready to be deployed in OSGi. |
| <code>camel-archetype-spring</code> | Creates a Camel project that loads up a CamelContext in Spring and defines a sample route in the Spring DSL. |
| <code>camel-archetype-spring-boot</code> | Creates a new Camel project using Spring Boot. |
| <code>camel-archetype-spring-dm</code> | Creates a Camel project that loads up a CamelContext in Spring and defines a sample route in the Spring DSL. Ready to be deployed in OSGi. |
| <code>camel-archetype-web</code> | Creates a Camel project that includes a few sample routes as a WAR file. |

Out of these 14 archetypes, the most commonly used one is probably the `camel-archetype-java` archetype. We'll try this out next.

USING THE CAMEL-ARCHETYPE-JAVA ARCHETYPE

The `camel-archetype-java` archetype listed in table 8.1 boots up a 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=2.17.3 \
-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 what you want

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 11.1 Layout of the project created by camel-archetype-java

```
order-router
.
├── pom.xml
└── ReadMe.txt
└── src
    ├── data
    │   └── message1.xml
    │   └── message2.xml
    ├── main
    │   ├── java
    │   │   └── camelinaction
    │   │       ├── MainApp.java
    │   │       └── MyRouteBuilder.java
    │   └── resources
    │       └── log4j.properties
    └── test
        ├── java
        │   └── camelinaction
        └── resources
```

- ① Test data
- ② CamelContext set up
- ③ Sample Java DSL route
- ④ Logging configuration

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 the context to stop.

While running, the sample route will consume files in the `src/data` directory and, based on the content, will route 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 start 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("cxf: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, of course, 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, we'll focus on adding component dependencies using Maven.

8.1.2 Camel Maven dependencies

Technically, Camel is just a Java application. To use it, you just need to 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. Back in chapter 6 you saw that this was accomplished by adding the following to the dependencies section of the POM file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>2.17.3</version>
</dependency>
```

This dependency element will tell Maven to download the `camel-ftp` JAR from Maven's central repository at <http://repo1.maven.org/maven2/org/apache/camel/camel-ftp/2.17.3/camel-ftp-2.17.3.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 36 transitive dependencies to our project! Luckily only 6 of them are needed at runtime; the other 30 are used during testing. The 6 transitive runtime dependencies can be viewed as a tree, as shown in figure 8.2.

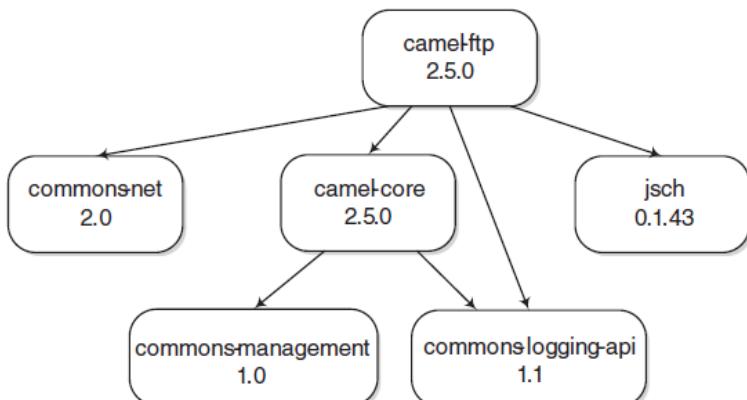


Figure 8.2 Transitive runtime dependencies of the camel-ftp module. When you add a dependency on camel-ftp to your project, you'll also get its transitive dependencies added to your classpath. In this case, commons-net, camel-core, and jsch are added. Additionally, camel-core has a dependency on jaxb-core, 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 actually 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 quite 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
```

After a few seconds of work, Maven will print out a listing like this:

```

camelinaction:chapter8-order-router:jar:2.0.0
+- org.apache.camel:camel-core:jar:2.17.2:compile
|   +- org.slf4j:slf4j-api:jar:1.7.12:compile
|   +- com.sun.xml.bind:jaxb-core:jar:2.2.11:compile
|   \- com.sun.xml.bind:jaxb-impl:jar:2.2.11:compile
+- org.apache.camel:camel-spring:jar:2.17.2:compile
|   +- org.springframework:spring-core:jar:4.2.6.RELEASE:compile
  
```

```

|   | \- commons-logging:commons-logging:jar:1.2:compile
|   +- org.springframework:spring-aop:jar:4.2.6.RELEASE:compile
|   | \- aopalliance:aopalliance:jar:1.0:compile
|   +- org.springframework:spring-context:jar:4.2.6.RELEASE:compile
|   +- org.springframework:spring-beans:jar:4.2.6.RELEASE:compile
|   +- org.springframework:spring-expression:jar:4.2.6.RELEASE:compile
|   \- org.springframework:spring-tx:jar:4.2.6.RELEASE:compile
+- org.apache.camel:camel-ftp:jar:2.17.2:compile
|   +- com.jcraft:jsch:jar:0.1.53:compile
|   \- commons-net:commons-net:jar:3.3:compile
+- log4j:log4j:jar:1.2.17:compile
\- org.slf4j:slf4j-log4j12:jar:1.7.12:compile

```

Here, you can see that Maven is adding 18 JARs to your project's compile-time classpath, even though you only added 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 a bit 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:

- 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/2.17.2/camel-ftp-2.17.2.pom>.
- Run mvn -f camel-ftp-2.17.2.pom dependency:copy-dependencies
- The dependencies for camel-ftp will be 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 will have to download the JARs yourself from the Maven central repo.

You now know all you need to develop Camel projects using Maven. To make you an even more productive Camel developer, let's now look at how you can develop Camel applications inside an IDE, like 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 Starting 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 will 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 recreate the chapter8-order-router example we looked at previously.

Click File > New > Maven Project to start up 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 just include "org.apache.camel", you can easily spot the Camel archetypes.

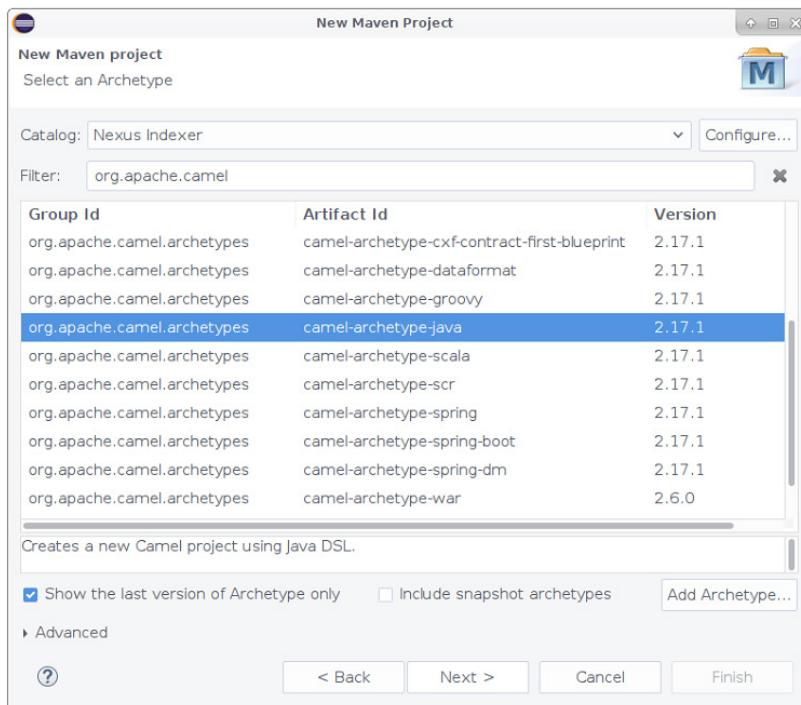


Figure 8.3 The New Maven Project wizard allows you to generate a new Camel project right in Eclipse.

Using Camel archetypes in this way is equivalent to the technique you used back in section 8.1.1.

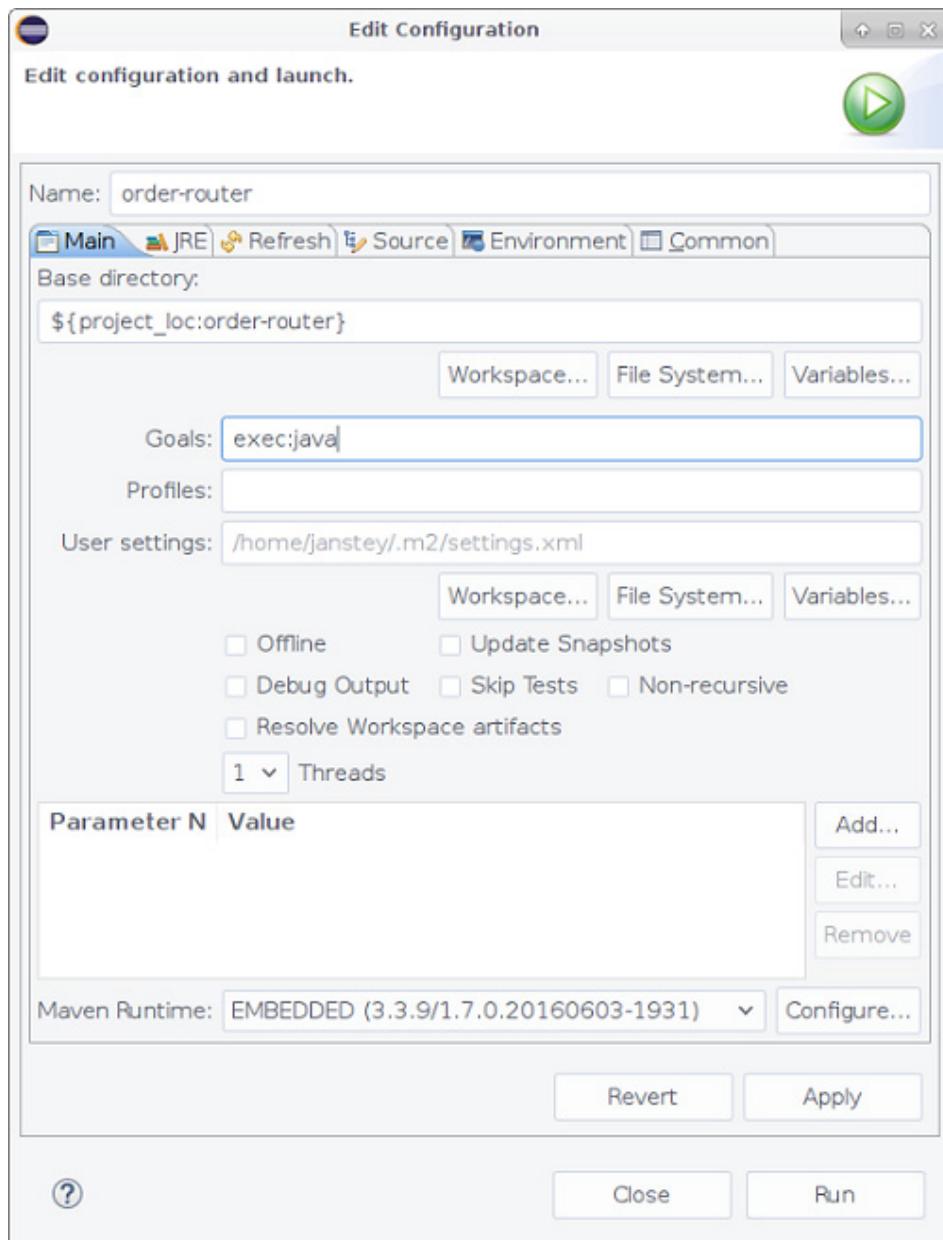
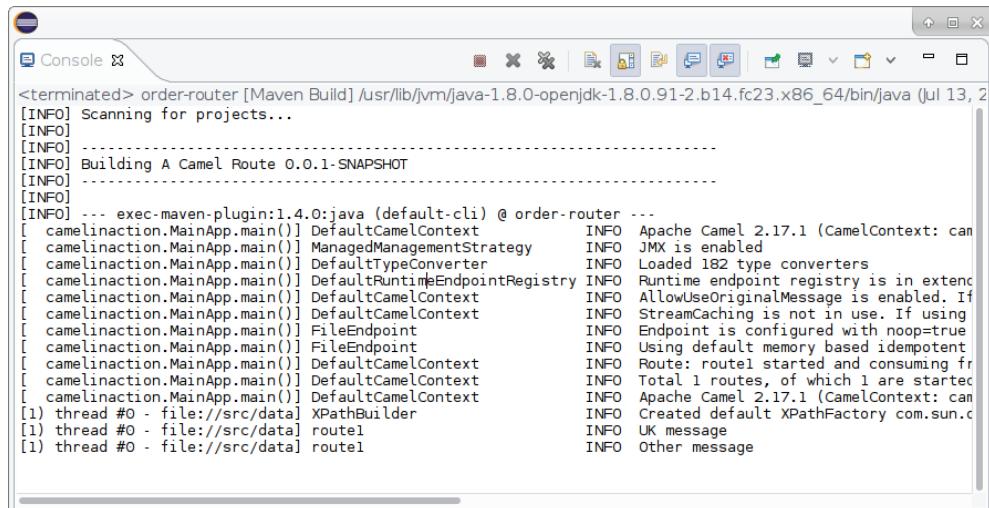


Figure 8.4 Right-clicking on the order-router project in the Package Explorer and clicking Run As > Maven Build will bring up the Edit Configuration dialog box shown here. The `exec:java` Maven goal has been entered.

To run the order-router project with Eclipse, right-click on the project in the Package Explorer and click Run As > Maven Build. This will bring 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.

Clicking Run in this dialog box will execute the `mvn exec:java` command in Eclipse, with console output showing in the Eclipse Console view.



The screenshot shows the Eclipse IDE's Console view. The title bar says "Console". The log output is as follows:

```

<terminated> order-router [Maven Build] /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.91-2-b14.fc23.x86_64/bin/java (Jul 13, 2017)
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building A Camel Route 0.0.1-SNAPSHOT
[INFO] -----
[INFO] [INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ order-router ---
[INFO] [camelinaction.MainApp.main()] DefaultCamelContext           INFO  Apache Camel 2.17.1 (CamelContext: camelContext)
[INFO] [camelinaction.MainApp.main()] ManagedManagementStrategy    INFO  JMX is enabled
[INFO] [camelinaction.MainApp.main()] DefaultTypeConverter        INFO  Loaded 182 type converters
[INFO] [camelinaction.MainApp.main()] DefaultRuntimeEndpointRegistry INFO  Runtime endpoint registry is in extended mode
[INFO] [camelinaction.MainApp.main()] DefaultCamelContext         INFO  AllowUseOriginalMessage is enabled. If using StreamCaching
[INFO] [camelinaction.MainApp.main()] DefaultCamelContext         INFO  StreamCaching is not in use. If using Endpoint is configured with noop=true
[INFO] [camelinaction.MainApp.main()] FileEndpoint                INFO  Using default memory based idempotent consumer
[INFO] [camelinaction.MainApp.main()] FileEndpoint                INFO  Route: routel started and consuming from file://src/data/routel
[INFO] [camelinaction.MainApp.main()] DefaultCamelContext         INFO  Total 1 routes, of which 1 are started
[INFO] [camelinaction.MainApp.main()] DefaultCamelContext         INFO  Apache Camel 2.17.1 (CamelContext: camelContext)
[INFO] [1] thread #0 - file://src/data/XPathBuilder              INFO  Created default XPathFactory com.sun.org.apache.xpath.lib.XPathFactoryImpl
[INFO] [1] thread #0 - file://src/data/routel                     INFO  UK message
[INFO] [1] thread #0 - file://src/data/routel                     INFO  Other message

```

Figure 8.5 Console output from the order-router project when running the `exec:java` Maven goal.

8.2.2 Debugging an issue with your new Camel project

So you've just created a new Camel project in Eclipse 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. However, you'll hit a couple of issues if you try and 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 we try and set a break point on say the `log("UK message")` line we won't actually hit that breakpoint for each message satisfying `when(xpath("/person/city = 'London'"))` – it will only break once on startup. This is because RouteBuilders just form the model from which Camel will create a graph of Processors from. The Processor graph forms the runtime structure that we can actually debug on a per message basis. One trick that is quick to use in a pinch is an anonymous Processor inside the route definition:

```
.when(xpath("/person/city = 'London'"))
    .log("UK message")
    .process(new Processor() {
        @Override
        public void process(Exchange arg0) throws Exception {
            // breakpoint goes here!
        }
    })
    .to("file:target/messages/uk")
```

This is of course not very helpful to use in the XML DSL and furthermore having to modify your route just to debug is not very nice. Often a better first approach is to not debug using the Eclipse debugger at all but use Camel's built in management and monitoring abilities. In chapter 16 we discuss these topics like:

- 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 – Of course logs are the first place you should go to when investigating a problem in most applications and Camel is no different. Section 16.3 covers this.
- Tracing – The tracer is probably the most useful tool Camel has builtin 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, there are a number of tooling projects out there to assist you with debugging Camel. In chapter 19 we cover:

- JBoss Tools for Apache Camel – JBoss Tools is an Eclipse plugin for Camel development which 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 very nice Camel plugin that allows you to step through a route visually in your browser!

DEBUGGING YOUR UNIT TESTS

Tooling projects like Hawtio and JBoss Tools 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, etc. 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. This means you can see what is happening before and after each processor invocation in a Camel route's runtime processor graph. Table 8.2 lists the Maven modules and also 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> |
| camel-test-karaf | <code>org.apache.camel.test.karaf.CamelKarafTestSupport</code> |
| camel-testng | <code>org.apache.camel.testng.CamelTestSupport</code> |

Once 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 is happening to the Exchange very precisely. You can also just run the test case and these debug methods will provide detailed tracing similar to Camel's tracer.

To try this out for yourself, you can run the example in the chapter8/order-router-full directory like:

```
mvn clean test -Dtest=OrderRouterDebuggerTest
```

Now you can say that you know how to debug a Camel application! Let's next discuss how you can start extending Camel itself by adding your own components.

8.3 Developing custom components

For most integration scenarios, there's a Camel component available to help. Sometimes, though, there's no Camel component 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, we'll look at how you can create your own custom component for Camel.

8.3.1 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 is what 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>). Just like 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 \
-DarchetypeGroupId=org.apache.camel.archetypes \
-DarchetypeArtifactId=camel-archetype-component \
-DarchetypeVersion=2.17.1 \
-DgroupId=camelinaction.component \
-DartifactId=custom \
-Dname=My \
-Dscheme=mycomponent
```

This will generate a project structure like that shown here.

Listing 11.2 Layout of a project created by camel-archetype-component

```

custom
└── pom.xml
└── ReadMe.txt
└── src
    ├── data
    └── main
        ├── java
        │   └── camelinaction
        │       └── component
        │           ├── MyComponent.java
        │           ├── MyConsumer.java
        │           ├── MyEndpoint.java
        │           └── MyProducer.java
        |
        └── resources
            └── META-INF
                └── services
                    └── org
                        └── apache
                            └── camel
                                └── component
                                    └── mycomponent      ②
    └── test
        ├── java
        │   └── camelinaction
        │       └── component
        │           └── MyComponentTest.java      ①
        └── resources
            └── log4j.properties

```

- ① Component implementation
- ② File that maps URI scheme to component class
- ① Test case for component

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.

8.3.2 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

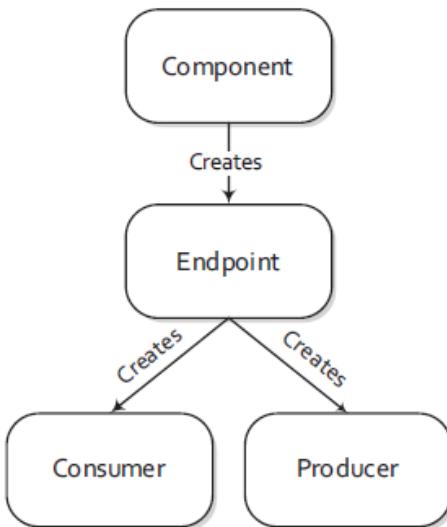


Figure 8.6 A Component creates an Endpoint, which then creates Producers and Consumers.

We'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 UriEndpointComponent class. The UriEndpointComponent class itself extends DefaultComponent and allows you to define the configurable parameters of the endpoint URI via annotations, which we'll touch on when we look at the implementation of MyEndpoint. The MyComponent class generated by the camel-archetype-component archetype forms a pretty 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.UriEndpointComponent;

/**
 * Represents the component that manages {@link MyEndpoint}.
 */
  
```

```

public class MyComponent extends UriEndpointComponent {

    public MyComponent() {
        super(MyEndpoint.class);
    }

    public MyComponent(CamelContext context) {
        super(context, MyEndpoint.class);
    }

    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 pretty 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 here.

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;

/**
 * Represents a My endpoint.
 */
@UriEndpoint(          ①
    scheme = "mycomponent",
    title = "My",
    syntax="mycomponent:name",
    consumerClass = MyConsumer.class,
    label = "My")
public class MyEndpoint extends DefaultEndpoint { ②
    @UriPath @Metadata(required = "true")          ③
    private String name;
    @UriParam(defaultValue = "10")                  ④
}

```

```

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
}

public Consumer createConsumer(Processor processor) throws Exception {
    return new MyConsumer(this, processor); 6
}

public boolean isSingleton() {
    return true;
}

/**
 * Some description of this option, and what it does
 */
public void setName(String name) {
    this.name = name;
}

public String getName() {
    return name;
}

/**
 * Some description of this option, and what it does
 */
public void setOption(int option) {
    this.option = option;
}

public int getOption() {
    return option;
}
}

```

- 1 Specifies that this endpoint is set up with annotations
- 2 Extends from default endpoint class
- 3 Specifies the URI path content
- 4 Specifies that this is a settable option on the endpoint URI
- 5 Creates new producer
- 6 Creates new consumer

The first thing you'll notice is that a lot of set up 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 the `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, etc. So if you want your component to be widely reused, it is important to add these annotations. However, they are 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`, is what tells Camel to expect that this endpoint is described via annotations ① . Here you can specify things like the scheme of the endpoint, its syntax, and any labels that you which add. Labels are most often used to categorize things in Camel. For example, if this component integrated with a messaging system you could add a label "messaging".

Like 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 very 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 we 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 like if its required or if it has a label associated with it ④ . `UriParam` specifies that this is a settable option on the endpoint URI. So with the code in Listing 8.3, you can have an option like:

```
mycomponent:endpointName?option=value1
```

As we mentioned in chapter 6, the `Endpoint` class acts as a factory for both consumers and producers. In this example, you're creating both producers ⑤ and consumers ⑥ , 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 only has a producer or consumer, not both. In that case, it's recommended that you throw an exception, so users know that 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.

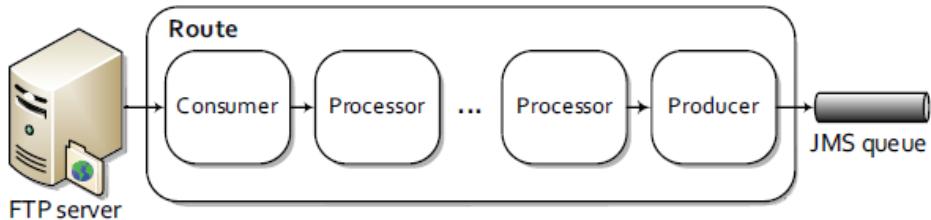


Figure 8.7 A simplified view of a route where the consumer and producer handle interfacing with external systems. Consumers take messages from an external system into Camel, and producers send messages to external systems.

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 this listing.

Listing 8.4 Custom Camel producer—`MyProducer`

```

package camelinaction.component;

import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultProducer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * The My producer.
 */
public class MyProducer extends DefaultProducer { 1
    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
        System.out.println(exchange.getIn().getBody()); 3
    }
}
  
```

1 Extends from default producer class

2 Serves as entry point to producer

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 a processor is, so the entry point into the producer is the `process` method. The sample component that was created automatically has a very 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 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 listing 8.5.

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;

/**
 * The My consumer.
 */
public class MyConsumer extends ScheduledPollConsumer { ①
    private final MyEndpoint endpoint;

    public MyConsumer(MyEndpoint endpoint, Processor processor) {
        super(endpoint, processor);
        this.endpoint = endpoint;
    }

    @Override
    protected int poll() throws Exception { ②
        Exchange exchange = endpoint.createExchange();

        // create a message body
        Date now = new Date();
        exchange.getIn().setBody("Hello World! The time is " + now);

        try {
            // send message to next processor in the route
            getProcessor().process(exchange); ③
            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());
    }
}
```

- 1 Extends from built-in consumer
 - 2 Is called every 500 ms
 - 1 Sends to next processor in route

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.

At this point, you should have a good understanding of what is required to create a new Camel component. You may even have a few ideas about what you'd like to bridge Camel to next! Next we'll discuss how to .

8.4 Generating components with the API component framework

So now you know how to write a component from a Camel API point of view. But what do you do when you actually get down and start interacting with other transports or APIs? Often the first major decision you will have to make is whether you will have to code all these interactions from scratch or if there is already a suitable library out there that you can reuse. Components in Camel's core module take the first approach if you are looking for inspiration for your implementation. Most of the other component modules take the latter approach and are backed by some 3rd party library.

Using a 3rd party library significantly reduces the size and complexity of the component since most of the grunt work is handled in the 3rd party library. Component development in this case becomes a pretty mundane task. You've seen the boiler plate code from a typical component in the previous section, now your task essentially becomes mapping elements of an endpoint URI to method calls in this 3rd party library.

For larger libraries this process can be quite daunting. Take for example the camel-box component, which integrates with the box.com file management service – it has over 50 operations accessible from the endpoint URI. Now that would have been a pain to implement. Dhiraj Bokde, a colleague of the authors, realized we could do so much better. So 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 how this can be done.

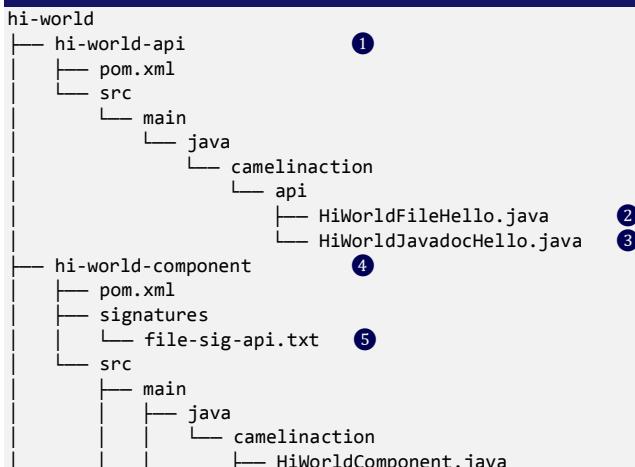
8.4.1 Generating the skeleton API project

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=2.17.1 \
-DgroupId=camelinaction \
-DartifactId=hi-world \
-Dname=HiWorld \
-Dscheme=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





- ① Project with sample APIs
- ② Sample API with method signatures defined in file
- ③ Sample API with method signatures defined in javadoc
- ④ Project containing the Camel component
- ⑤ Method signatures for API ②

The generated multi-module Maven project is a bit bigger than the custom component we 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 only need to keep a portion of the `hi-world-component` project ④ .

The `hi-world-api` project ① is the sample 3rd party API that our Camel component ④ will be using. In a real Camel API component instead of this sample API project, you'd be pointing the tooling at some already released library. For example, in the `camel-box` component (part of Apache Camel), a dependency is on an actual 3rd party library:

```
<dependency>
  <groupId>net.box</groupId>
  <artifactId>boxjavalibv2</artifactId>
</dependency>
```

whereas our project will instead depend on the sample API project:

```
<dependency>
  <groupId>camelinaction</groupId>
  <artifactId>hi-world-api</artifactId>
</dependency>
```

Lets looks at how we can generate a working Camel component from this API.

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

In Listing 8.6 we saw how there were two projects listed: a sample API ① and a Camel component using that sample API ④. Looking at the pom.xml for the Camel component we first have a dependency on the sample API as follows:

```
<dependency>
    <groupId>camelinaction</groupId>
    <artifactId>hi-world-api</artifactId>
</dependency>
<dependency>
    <groupId>camelinaction</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 are not. So in order to get the full set of names (i.e. 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. Listing 8.7 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
                        <proxyClass>camelinaction.api.HiWorldFileHello</proxyClass> ②
                        <fromSignatureFile>signatures/file-sig-api.txt</fromSignatureFile> ③
                    </api>
                    <api>
                        <apiName>hello-javadoc</apiName> ④
                        <proxyClass>camelinaction.api.HiWorldJavadocHello</proxyClass> ⑤
                        <fromJavadoc/> ⑥
                    </api>
                </apis>
            </configuration>
        </execution>
    </executions>
</plugin>
```

- 1 API name for file based signature
- 2 Class to interrogate using reflection
- 3 Location of text file with method signatures for class 2
- 4 API name for javadoc based signature
- 5 Class to interrogate using reflection
- 6 Specify to check for method signatures in javadoc for class defined in 5

Here we can see the Maven plugin is configured to generate an endpoint URI to Java API mapping for two classes 2 5 . Let's first take a look at the file-supplemented API camelinaction.api.HiWorldFileHello shown in Listing 8.8 below:

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;
    }
}
```

So when the Maven plugin interrogates `HiWorldFileHello`, it finds 3 methods and determines that 3 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 (i.e. `greetMe`, `greetUs`, and `sayHi`) come from the methods in the `HiWorldFileHello` class.

Input data is handled differently depending on the number of parameters. So, 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. So, a URI like:

```
hiworld://hello-file/greetMe?inBody=name
```

Will essentially be calling:

```
HiWorldFileHello.greetMe(/* Camel message body */)
```

For methods with more than one parameter, each parameter is mapped to a message header. So, for the greetUs method, you'd set up headers like:

```
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:

1. Camel is always used for the prefix,
2. The HiWorld portion of the URI comes from the `-Dname=HiWorld` argument passed into the Maven archetype plugin in section 8.4.1,
3. Finally, there is a dot follow by the parameter name.

Recall that for this API we are 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 below:

```
public String sayHi();
public String greetMe(String name);
public String greetUs(String name1, String name2);
```

Without this file we'd just have parameter names like `arg0`, `arg1`, etc.

This process is even easier in the case where you have javadoc available for the 3rd party API. We tried the javadoc way of providing parameter names in Listing 8.7 ⑥ . The proxy class which we are using `- camelinaction.api.HiWorldJavadocHello` is essentially the same as `camelinaction.api.HiWorldFileHello` shown in Listing 8.8. It produces 3 sub URI schemes like:

```
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 hand crafted APIs, there isn't much we need to change in the conversion over to Camel endpoints. For larger APIs though, customization is a likely requirement. There are a ton of customization options available to address this so let's go over them.

8.4.3 Advanced configuration options

There are times when you may need to customize the mapping from 3rd party API to 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 also the naming used on the remote API has conflicts with other names you want to use in your component. Or perhaps you think you can name things better.

Let's take a look at some of the advanced configuration options you have available for both file and javadoc supplemented API components.

GLOBAL CONFIGURATION OPTIONS

There are six configuration options you can apply to both javadoc supplemented APIs and file signature file supplemented 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 3rd-party API. For example, in our API the `greetMe` method has a single parameter called “name”. Perhaps it would be more useful in our 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>
        <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, etc. You can also use a regular expression to grab text out of the parameter type as well. To do this you set the `replaceWithType` element to true and specific a regular expression in the `argType` element. The follow example will change the `greetMe` method “name” parameter to “usernameString”:

```
<substitutions>
  <substitution>
    <method>^greetMe$</method>
    <argName>^name$</argName>
```

```

<argType>^java.lang.(.+)$</argType>
<replacement>username$1</replacement>
<replaceWith>true</replaceWith>
</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 our 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 `methodAlias` element as `$1`, `$2`, and so on. In this case we have made the following 2 URIs equivalent:

```

hiworld://hello-javadoc/greetUs
hiworld://hello-javadoc/us
```

As well as 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 3rd 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>
      <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 were not in the 3rd party API. For example, let's add a “language” option:

```

<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>
```

JAVADOC ONLY CONFIGURATION OPTIONS

There are some additional options that are only available within the `fromJavadoc` element. Let's take a look at them:

- `excludePackages` – Exclude methods from proxy classes with specified package name. Useful for filtering out methods from unwanted super classes. Default excludes are methods from any classes in `java.lang` package.

```
<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 (i.e. unwanted super classes) matching specified name.

```
<fromJavadoc>
  <excludeClasses>SomeUnwantedAbstractSuperClass</excludeClasses>
</fromJavadoc>
```

- `includeMethods` – Only methods matching this pattern will be included.

```
<fromJavadoc>
  <includeMethods>greetMe</includeMethods>
</fromJavadoc>
```

Only include the `greetMe` method in processing.

- `excludeMethods` – Any methods matching this pattern will not be included.

```
<fromJavadoc>
  <excludeMethods>greetMe</excludeMethods>
</fromJavadoc>
```

Do not include the `greetMe` method in processing.

- `includeStaticMethods` – Also include static methods. This is false by default.

```
<fromJavadoc>
<includeStaticMethods>true</includeStaticMethods>
</fromJavadoc>
```

8.4.4 Implementing remaining functionality

So with the sample component generated from `camel-archetype-api-component`, there isn't much left to do. However, if we were running this against a real 3rd party API, we would 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 the `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 our case we have a test case for the javadoc and file supplemented APIs in the `target/generated-test-sources/camel-component/camelinaction` directory:

1. `HiWorldFileHelloIntegrationTest.java`
2. `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 our routes look like:

```
@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 how there isn't a route with a `hi-world consumer` (i.e. no `from("hiworld...")` - you'll have to fill that in yourself if the component will support it. There is a test case created for each API method. For the `greetUs` method the test case looks like:

```
@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 of course, but it is quite handy because it shows you how many parameters to set and also what their names are. When you've filled in some data and also some additional asserts, you can remove the `@Ignore` annotation.

FILLING IN THE OTHER BLANKS

Now that we have seen how the tests are set up, let's take a look at the actual component. Recall how the sample `hi-world-component` contained 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 here would be `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 we 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>camelinaaction.api.HiWorldFileHello</proxyClass>
  <fromSignatureFile>signatures/file-sig-api.txt</fromSignatureFile>
</api>
<api>
  <apiName>hello-javadoc</apiName>
  <proxyClass>camelinaaction.api.HiWorldJavadocHello</proxyClass>
  <fromJavadoc/>
</api>
```

So if this matches up, what's left to do? Well, in many cases the Java library containing this proxy class needs to be configured. So this could be things like authentication for a remote server, opening a session, setting connection properties, etc. These are things that often differ for each 3rd party library and as such 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 that are supported by the API component framework. These are:

- camel-linkedin
 - camel-braintree
 - camel-google-mail
 - camel-box
 - camel-google-calendar
 - camel-google-drive
 - camel-olingo2

By now you should have a very good understanding of how to write a new component for Apache Camel. Hopefully you'll also be considering contributing³ them back to the project! Let's now look at another extension point in Camel: data formats.

³ Appendix D covers contributing to Apache Camel.

8.5 Developing data formats

As we saw back 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, 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 different data formats but there may be times when you need to write your own. In this section, we'll look at how you can develop a data format that can reverse strings. Let's first look at how to set up the initial boiler plate code using an archetype.

8.5.1 Generating the skeleton data format 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 following Maven command:

```
mvn archetype:generate \
-B \
-DarchetypeGroupId=org.apache.camel.archetypes \
-DarchetypeArtifactId=camel-archetype-dataformat \
-DarchetypeVersion=2.17.1 \
-DgroupId=camelinaction \
-DartifactId=reverse-dataformat \
-Dname=Reverse \
-Dscheme=reverse
```

The resultant `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
    │   └── resources
    │       └── META-INF
    │           └── services
    │               └── org
    │                   └── apache
    │                       └── camel
    │                           └── dataformat
    │                               └── reverse
    └── test
        └── java
            └── camelinaction
```

```

    └── ReverseDataFormatTest.java ①
└── resources
    └── log4j.properties

```

- ① Skeleton data format class
- ② Ensure custom data format is registered with “reverse” name in Camel
- ③ Test case for data format

This data format is actually usable right away and even has a working test case. However, it doesn't do much as it just returns the data unmodified. So let's look into how we can make our data format actually change the data.

8.5.2 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 you could implement a string-reversing data format.

Listing 8.10 Developing a custom data format that can reverse strings

```

package camelinaction;

import java.io.InputStream;
import java.io.OutputStream;

import org.apache.camel.Exchange;
import org.apache.camel.spi.DataFormat;
import org.apache.camel.spi.DataFormatName;
import org.apache.camel.support.ServiceSupport,

/**
 * A <a href="http://camel.apache.org/data-format.html">data format</a> ({@link DataFormat})
 * for Reverse data.
 */
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
        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
        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
}

}

```

- ① Marshals to reverse string
- ② Unmarshals to unreverse string

The custom data format must implement the `DataFormat` interface, which forces you to develop two methods: `marshal` ① and `unmarshal` ② . That's no surprise, as they're the same methods you use in the route. The `marshal` method ① needs to output the result to the `OutputStream`. To do this, you need to get the message payload as a `byte[]`, and then reverse it with a helper method. Then you write that data to the `OutputStream`. Note that you use the Camel type converters to return the message payload as a `byte[]`. This is very 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. We cover Camel's type converters in section 3.6.

To use this new data format in a route, all you have to do is define it as a bean and refer to it 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>
    </route>

```

```

<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 this looks like:

```

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 like:

```

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 understanding of what is required to create a new Camel data format. Another useful way of extending Camel is by writing custom interceptors. Let's look at this next.

8.6 Developing interceptors

Interceptors in Camel are used to perform some action on a message as it goes in and out of a processor. Features like the tracer discussed in chapter 16 use a custom interceptor to trace each message going in and out of processors. We also talked about interceptors back in chapter 6, where you used them to simulate errors occurring on a particular endpoint. Convenience methods built into Camel's DSL were used in that case.

In this section, we'll look at how you can create your own custom interceptor.

8.6.1 Creating an InterceptStrategy

To create a new interceptor, you need to use the `InterceptStrategy` interface. By implementing a custom `InterceptStrategy`, you gain complete control over what the interceptor does.

The `InterceptStrategy` interface only has a single method:

```
Processor wrapProcessorInInterceptors(
    CamelContext context,
    ProcessorDefinition<?> definition,
    Processor target,
    Processor nextTarget) throws Exception;
```

This method essentially wraps each processor within a route with another processor. This wrapper processor will contain the logic you want for your interceptor.

In Camel, `InterceptStrategy` classes are used to implement a delay after each node in a route, to trace messages as they flow through a route, and to record performance metrics. Figure 8.8 shows a conceptual view of how an `InterceptStrategy` modifies a route.

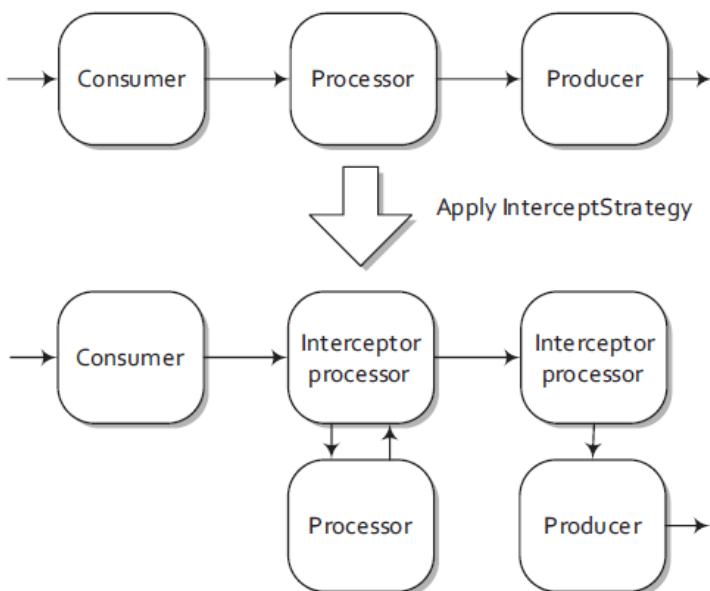


Figure 8.8 Applying an `InterceptStrategy` to a route essentially wraps each processor with an interceptor processor.

Every processor in the route shown in figure 8.8 is wrapped by an interceptor processor. This work of modifying the route is done automatically when you add the `InterceptStrategy` to the `CamelContext`. You can add an `InterceptStrategy` directly to the `CamelContext` with a single method call:

```
context.addInterceptStrategy(new MyInterceptor());
```

Adding an `InterceptStrategy` in XML is also easy. You just add one as a bean, and Camel will automatically find it on startup and add it to the `CamelContext`. This is all you would need to write:

```
<bean id="myInterceptor" class="camelinaction.MyInterceptor"/>
```

What happens if you define more than one strategy? Figure 8.9 shows that in this case Camel will stack the interceptors used for each real processor.

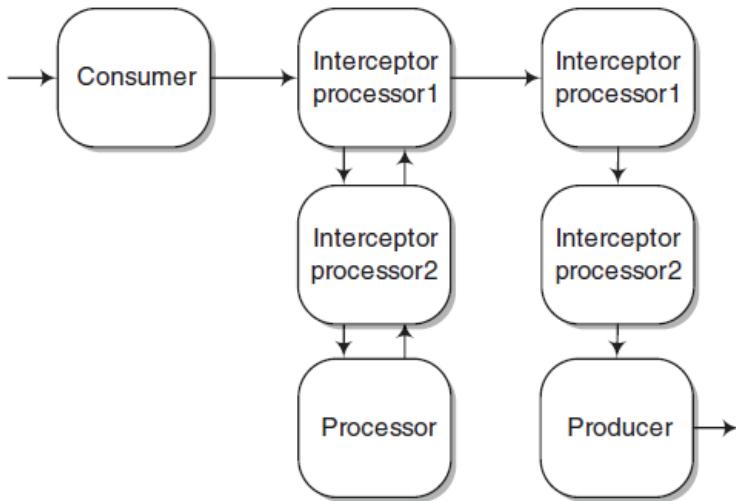


Figure 8.9 Interceptors are stackable, meaning the more `InterceptStrategy` classes you add to the `CamelContext`, the more interceptor processors will be added before each real processor is executed.

You may be wondering what an actual interceptor looks like. In the simplest case, you could just log an entry message before the processor and an exit message after the processor has completed. This is demonstrated in listing 8.11.

Listing 8.11 A custom interceptor

```

public class MyInterceptor implements InterceptStrategy {
    private static final transient Log log
        = LogFactory.getLog(MyInterceptor.class);
    public Processor wrapProcessorInInterceptors(
        CamelContext context,
        ProcessorDefinition<?> definition,
        final Processor target,
        Processor nextTarget)
        throws Exception {
        return new DelegateAsyncProcessor(new Processor() {
            public void process(Exchange exchange) throws Exception {
                log.info("Entering the processor...");
                target.process(exchange);
                log.info("Exiting the processor...");
            }
        });
    }
}
  
```

①

②

③

- 1 The target processor model
- 2 The target processor
- 3 Call into the target processor

The `wrapProcessorInInterceptors` method gives the interceptor developer plenty of information to help create a custom interceptor. First off, you have access to the entire `CamelContext`, which lets you access many different services, like the bean registry or the many type converters. Next, and most importantly, you have access to the target processor **2** that you need to call, and its DSL object equivalent **1**. Why was the DSL object provided? The DSL object provides richer description details than the runtime processor, so if your interceptor is, for example, graphically reporting the structure of a Camel route as it's being used, more detailed descriptions of each node may be helpful.

It isn't enforced by the interface, but interceptors are supposed to call the target processor **3** when they have done their work. After the target processor is finished, the interceptor can do some more work before returning to the next node in the route. In this way, an interceptor allows you to do operations before and after each processor in a route. Sometimes, you don't want to attach an interceptor to a particular processor. In this case, instead of always returning a wrapped processor, you could return the target processor. This would leave the target processor out of the interceptor scheme.

For example, you could ignore `wireTap` nodes from the interceptor scheme by using information from the processor's model class:

```
if ("wireTap".equals(definition.getShortName())) {
    return target;
} else {
    return new Processor() {
        ...
    };
}
```

This checks the name of the processor through its DSL object, and if it's a wire tap, it doesn't wrap it with an interceptor.

8.7 Summary and best practices

Knowing how to create Camel projects is very 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 what cool Camel applications you can create, having read about the features first. At this point, though, you should be well-equipped to start your own Camel application and make it do useful work.

Before we move on, there are a few key ideas that you should 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 that are provided by Camel will get you started much faster.

- *The easiest way to manage Camel library dependencies is with Maven.* 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 make Camel development easier.* From auto completion 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 are 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 like Hawtio or JBoss Tools 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 3rd 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. Like for components, Camel allows you to write and load up your own custom data formats easily. There's also a Maven archetype for starting a custom data format project.*
- *Use interceptors to inject processing around nodes in a route.* It's true, Camel interceptors are an advanced topic. They give you the additional power to control or monitor what's happening inside a route.

In the next chapter, we'll look at a topic that can help make you a successful integration specialist, and without it, you'll almost certainly be in trouble: testing with Camel. We'll also look at how you can simulate 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 will often involve a myriad of systems ranging from legacy systems, closed source commercial products, home grown applications, and so on. These systems often have no built-in support for automated testing.

The poor folks who are tasked to test such integrated systems often have no other choice to play through manual use-cases which involves triggering events from one window/terminal, and then crossing 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 then they would have to *reset the system* to ensures in correct state before caring on with the testing.

During development the systems being integrated are often tests systems, from other teams, or only available in production. If the systems are not network connected, then internal company procedures may hold back a speedy resolution. With the higher number of systems involved, the more difficult it becomes for developers to build and test their work.

Does this sound familiar? It should because these problems have been around for decades and therefore people have found out ways around this to let them 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 with what is the Camel test-kit and how to start doing Camel testing with different runtime platforms such as standalone Java, Spring-Boot, CDI, OSGi and JEE servers such as WildFly.

So far into this book you should have 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 will dive deeper into what the mock component is, and what functionality it offers for testing and how you can get more out of it in your tests.

We have said it before: integration is hard. It would be easier to build and integrate systems if nothing ever went wrong. But what about when things go wrong, how do you test your systems when a remote system is unavailable, when invalid data is returned, or a database starts failing with SQL violation errors. This chapter covers how you can simulate such error conditions in your tests.

Integration systems only start to add value to your business 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 will learn what strategies the Camel test-kit offers for testing Camel routes that are built as production ready routes.

At the end of the chapter we look at three third party testing frameworks for doing integration testing and see how you can use them with your Camel applications.

There are more testing disciplines which we do not cover in this book but are worth keeping in mind such as load, performance, and stress testing. In recent time companies have started seeing the benefits of having a continuous integration and deployment platform that helps drive test automation and makes your organization more agile and production. All of this is great topics that are part of a great good that are not integration style projects and therefore out of the scope of this book.

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.

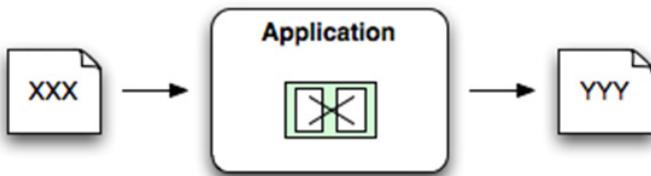


Figure 9.1 Testing a Camel application by sending a message to the application and then verifying the returned output

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.

This chapter is the longest chapter in the book so prepare yourself for a long journey. We wanted to provide you with a lot of details how to get successful with testing your Camel based applications. For example in section 9.2 we are covering six different ways of testing Camel using different runtime platforms. If you are not going to use some of these platforms such as JEE, or OSGi then you can skip those sections. However we put a lot of energy and effort into this book, so we hope to see you all the way to the finish line.

Enough talk let's get started.

9.1 Introducing the Camel Test Kit

Camel provides rich facilities for testing your projects, and it includes a test kit that gets you writing unit tests quickly in familiar waters using the regular JUnit API. In fact, it's the same test kit that Camel uses for testing itself. Figure 9.2 gives a high-level overview of the Camel Test Kit.

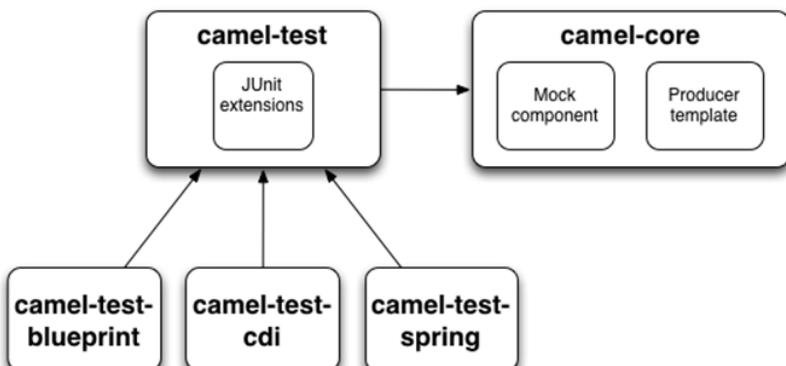


Figure 9.2 The Camel Test Kit is provided camel-test JAR that contains the JUnit extensions for unit testing, and three specialized JARs for testing with OSGi Blueprint, CDI, and Spring. The Camel Test Kit uses the Mock component and ProducerTemplate from camel-core JAR.

Figure 9.2 boils down to four parts. The camel-test JAR is the core test module includes the JUnit extensions as a number of classes on top of JUnit that make unit testing with Camel much easier. We'll 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, and for CDI testing use camel-test-cdi, and for Camel with Spring use camel-test-spring. In this section we will focus on testing standalone Camel applications using plain Java. In section 9.2 we will cover testing Camel with Spring, OSGi Blueprint and CDI.

The Camel Test Kit uses the Mock component from camel-core which is covered in section 9.3. And you're already familiar with the ProducerTemplate—it's a convenient feature that allows you to easily send messages to Camel when testing.

Let's now look at the Camel JUnit extensions and see how to use them to write Camel unit tests.

9.1.1 The Camel JUnit extensions

So 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 there are three which are intended to be used by end users and 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 |
|---|--|
| org.apache.camel.test.junit4.TestSupport | Abstract base class with additional builder and assertion methods. |
| org.apache.camel.test.junit4.CamelTestSupport | Base test class prepared for testing Camel routes. This is the test class you should use when testing your Camel routes. |
| org.apache.camel.test.AvailablePortFinder | 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 we'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. So 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 very 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—it's the easiest to get started 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>2.17.1</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 listing 9.1.

Listing 6.1 A first unit test using the Camel Test Kit

```
package camelinaction;

import java.io.File;
import org.apache.camel.Exchange;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.test.junit4.CamelTestSupport;
import org.junit.Test;

public class FirstTest extends CamelTestSupport {

    @Override
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            public void configure() throws Exception {           ①
                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

        Thread.sleep(2000);

        File target = new File("target/outbox/hello.txt");
        assertTrue("File not moved", target.exists());      ③
    }
}
```

- ① Defines route to test
- ② Creates hello.txt file
- ③ Verifies file is moved

The `FirstTest` class must extend the `org.apache.camel.junit4.CamelTestSupport` class to conveniently leverage 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 leverages 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 one second 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 two 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 it's what you expect. You may remember that Camel provides a very 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 is no need to fiddle with the various cumbersome Java IO 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 the `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());           ①

    String content = context.getTypeConverter()
        .convertTo(String.class, target);
    assertEquals("Hello World", content);                     ②
}

```

- ① Assert the file is moved
- ② Assert the file content is Hello World

Is there one thing in the test code that annoys you? Yeah the `Thread.sleep(2000)` to wait for Camel to pickup and process the file. Yes Camel can of course do better, and we can use something called `NotifyBuilder` to setup a pre-condition using the builder, and then let it wait until the condition is satisfied before continuing on. As we will cover `NotifyBuilder` later in section 9.4.2 we will just quickly show you the revised code:

```

@Test
public void testMoveFile() throws Exception {
    NotifyBuilder notify = new NotifyBuilder(context)
        .whenDone(1).create();          ①

    template.sendBodyAndHeader("file://target/inbox", "Hello World",
        Exchange.FILE_NAME, "hello.txt");

    assertTrue(notify.matchesMockWaitTime());                  ②

    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());            ③

    String content = context.getTypeConverter()
        .convertTo(String.class, target);
    assertEquals("Hello World", content);                     ④
}

```

- ① Setup pre-condition on `NotifyBuilder` to be when one message is done
- ② `NotifyBuilder` is waiting for the pre-condition to be satisfied
- ③ Assert the file is moved
- ④ Assert the file content is Hello World

The book's source code includes this revised example. You can try it on your own by running the following Maven goal from the chapter9/java directory:

```
mvn test -Dtest=FirstNoSleepTest
```

The preceding examples cover the case where 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

It's common to define Camel routes in separate `RouteBuilder` classes, 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 quite 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.

The accompanying source code of the book contains this example in the `chapter9/java` directory which you can try using the following Maven goal:

```
mvn test -Dtest=ExistingRouteBuilderTest
```

Notice that the `FileMoveRoute` class is located in the `src/main/java/camelinaction` directory and is not located in the test directory.

Now you have 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. However Camel can run in many different platforms. So lets dive into covering the most popular choices.

9.2 Testing Camel with Spring, Blueprint, and CDI

Camel applications are being used in many different environments, and together with various frameworks. Other the years people often run Camel together with either Spring, OSGi Blueprint or in more recently using CDI with JEE applications.

In this section we will demonstrate how to get started with 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/JBoss Fuse
- Camel using CDI running standalone using JBoss Weld CDI container
- Camel using CDI running in WildFly

We will in the following six sections cover each bullet item from the list. It is worth emphasizing that the goal for this section is to show you how to get on a good start testing Camel on those six different platforms. The focus is therefore on using a simple test case as example and highlight how the testing is specific to the chosen platform. We will dive deeper into testing topics such as integration and systems test, and more elaborate test functionality from Camel in the sections to follow. But first lets get you on a good start first.

The next section shows how to test using Spring XML-based routes.

9.2.1 Camel testing with Spring XML

You use camel-test-spring to test Spring based Camel applications. In the Maven pom.xml add the following dependency:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test-spring</artifactId>
  <version>2.17.1</version>
  <scope>test</scope>
</dependency>
```

In this section we'll look at unit testing the route in listing 9.2.

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 etc.

The camel-test-spring module is an extension to camel-test which means all the test principles you learned in section 9.1 also applies. 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 listing 9.3 illustrates this point.

Listing 9.3 A first unit test using Spring XML routes

```
package camelinaction;

import java.io.File;
import org.apache.camel.Exchange;
import org.apache.camel.test.junit4.CamelSpringTestSupport;
import org.junit.Test;
import org.springframework.context.support.AbstractXmlApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringFirstTest extends CamelSpringTestSupport {

    protected AbstractXmlApplicationContext createApplicationContext() {
        return new ClassPathXmlApplicationContext(
            "camelinaction/firststep.xml"); ①
    }

    @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);
    }
}
```

① Loads Spring XML file

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 in when testing using Java DSL from section 9.1, which means you can use the same unit testing code regardless of how the route is defined.

The source code for the book contains this example in the `chapter9/spring-xml` directory which you can try using the following Maven goal:

```
mvn test -Dtest=SpringFirstTest
```

UNIT TESTING WITH @RUNWITH

The test class in listing 9.3 was extending the `CamelSpringTestSupport` class. However you can also write Camel unit tests without extending a base class which is more modern style with JUnit. Listing 9.4 shows how to write the unit test in a more modern style.

Listing 9.4 Using modern JUnit style to write a Camel Spring unit test

```
package camelinaction;

import java.io.File;
import org.apache.camel.CamelContext;
import org.apache.camel.Exchange;
import org.apache.camel.ProducerTemplate;
import org.apache.camel.test.spring.CamelSpringRunner;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

@RunWith(CamelSpringRunner.class) ①
@ContextConfiguration(locations = {"firststep.xml"}) ②
public class SpringFirstRunWithTest {

    @Autowired
    private CamelContext context; ③
    @Autowired
    private ProducerTemplate template; ④

    @Test
    public void testMoveFile() throws Exception { ⑤
        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);
    }
}
```

- ① Run the test using Camel Spring runner
- ② Declare the location of the Spring XML file relative to this unit test
- ③ Dependency inject CamelContext
- ④ Dependency inject ProducerTemplate
- ⑤ Test method same as in listing 9.3.

Instead of extending a base class the test class is annotated with `@RunWith(CamelSpringRunner.class)` ① . The `CamelSpringRunner` is an extension to Spring's `SpringJUnit4ClassRunner` that integrates Camel with Spring and allows to use additional Camel annotations in the test. We will 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. Notice the location is relative to the current class, so `firststep.xml` file is also located in the `camelinaction` package.

To use Camel during the unit test we need to dependency inject both `CamelContext` ③ and `ProducerTemplate` ④ . The test method is unchanged ⑤ .

This example is also provided with the source code in the `chapter9/spring-xml` directory which you can try by running the following Maven goal:

```
mvn test -Dtest=SpringFirstRunWithTest
```

Now lets move on to testing Spring application that are using Java Config instead of XML.

9.2.2 Camel testing with Spring Java Config

The Spring framework started mainly using XML configuration files which became a very popular choice of building applications. Camel works very well with XML by offering the XML DSL so you can define Camel routes directly in your Spring XML files. However in recent time an alternative configuration using plain Java has become popular as well - with a lot of thanks to the rising popularity of Spring Boot.

You can use Spring Java Config both standalone and with Spring. We will cover both scenarios in this and the following section.

The example we use is the `FileMoveRoute` as shown:

```
package camelinaction;

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

@Component ①
public class FileMoveRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("file://target/inbox")
            .to("file://target/outbox");
    }
}
```

- ① Annotate the class with `@Component` to make Spring discover the class easily

The `FileMoveRoute` class is a plain Camel route which has been annotated with `@Component` ① . This enables Spring Java Config to discover the route during startup and automatic create an instance which in turn Camel discovers from Spring and automatic includes the route.

To bootstrap and run a Camel with Spring Java Config you need:

- A Java main class to start the application
- A @Configuration class to bootstrap Spring Java Config

You can combine both in the same class as shown in listing 9.5.

Listing 9.5 - Java main class to bootstrap Spring Java Config with Camel

```
package camelinaction;

import org.apache.camel.spring.javaconfig.CamelConfiguration;
import org.apache.camel.spring.javaconfig.Main;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("camelinaction")          ①
public class MyApplication extends CamelConfiguration {      ②

    public static void main(String[] args) throws Exception {
        Main main = new Main();           ④
        main.setConfigClass(MyApplication.class);  ⑤
        main.run();
    }
}
```

- ① Marks this class as a Spring Java configuration class
- ② Tell Spring to scan in the package for classes with Spring annotations
- ③ To automatic enable Camel
- ④ Main class to easily start the application and keep the JVM running
- ⑤ Use this class as the configuration class

The `MyApplication` class extends `CamelConfiguration` ③ and has been marked as a `@Configuration` ① class. This allows us to configure the application in this class using Spring Java Config. However this example is very simple and there are no additional configuration.

TIP You can add methods annotated with `@Bean` in the configuration class to have Spring automatic call the methods and enlist the returned bean instances into the Spring bean context.

To include the Camel routes we have enabled `@ComponentScan` ③ to scan for classes that has been annotated with `@Component` which would automatic be enlisted into the Spring bean context. Which in turn allows Camel to discover the routes from Spring and automatic include the route when Camel startup.

To make it easy to run this application standalone we use a `Main` class from `camel-spring-java-config` that in a few lines of code startup the application and keep the JVM running ④ ⑤

So how do we unit test this application?

You can test the application using the same `CamelSpringTestSupport` we used when testing Camel with Spring XML files. The difference is how you create Spring in the `createApplicationContext` method as shown in listing 9.6.

Listing 9.6 Testing Spring Java Config application using CamelSpringTestSupport

```

import java.io.File;
import org.apache.camel.Exchange;
import org.apache.camel.test.junit4.CamelSpringTestSupport;
import org.junit.Test;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringFirstTest extends CamelSpringTestSupport {

    @Override
    protected AbstractApplicationContext createApplicationContext() {
        AnnotationConfigApplicationContext acc = new AnnotationConfigApplicationContext(); ①
        acc.register(MyApplication.class); ②
        return acc;
    }

    @Test
    public void testMoveFile() throws Exception { 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);
    }
}

```

- ① Creates a Spring Java Config context
- ② Register the @Configuration class in the Spring context
- ③ The unit test method is identical with testing using Spring XML files

The main difference is that Spring Java Config uses `AnnotationConfigApplicationContext`

- ① as the Spring bean container. You then have to configure which `@Configuration` class to use using the `register` method ②. The rest of the unit test is identical to listing 9.3 where we were testing using Spring XML.

The source code for the book contains the example in the `chapter9/spring-java-config` directory which you can try using

```
mvn test -Dtest=FirstTest
```

You can also unit test without extending `CamelSpringTestSupport` class, but instead use a set of annotations. This has become more popular in recent time 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. Listing 9.7 shows how to do that.

Listing 9.7 Unit testing with @RunWith instead of extending CamelSpringTestSupport class

```
import java.io.File;
import org.apache.camel.Exchange;
import org.apache.camel.test.spring.CamelSpringDelegatingTestContextLoader;
import org.apache.camel.test.spring.CamelSpringRunner;
import org.junit.Test;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;

@RunWith(CamelSpringRunner.class)                      ①
@ContextConfiguration(                                ②
    classes = MyApplication.class,
    loader = CamelSpringDelegatingTestContextLoader.class)
public class RuntWithFirstTest {

    @Autowired
    private CamelContext context;                      ③
    @Autowired
    private ProducerTemplate template;                 ③

    @Test
    public void testMoveFile() throws Exception { ④
        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);
    }
}
```

- ① Use `@RunWith` with the `CamelSpring` runner
- ② Configure Spring to use the `@Configuration` class
- ③ Dependency inject `CamelContext` and `ProducerTemplate`
- ④ The unit test method is identical with testing without using `@RunWith`

The unit test uses `CamelSpringRunner` as the `@RunWith` runner ①. To configure Spring we are using `@ContextConfiguration` ② here we refer to the class that holds the `@Configuration`. Then the loader refers to `CamelSpringDelegatingTestContextLoader` which is a class that enables using a set of additional annotation to enable certain features

during testing. However in this simple example we are not using any of these features and you can omit declaring the loader. But its a good habit to get used to it so you are ready for using Camel testing capabilities such as auto mocking, and advice. All capabilities that are revealed later in this chapter.

Then we dependency inject `CamelContext` and `ProducerTemplate` ③ which we use in the unit test method ④ . You have to do this because the test class no longer extends `CamelTestSpringSupport` which provides the `CamelContext` and `ProducerTemplate` automatically.

You can try this example with the book's source code in the `chapter9/spring-java-config` directory using the following Maven goal:

```
mvn test -Dtest=RunWithFirstTest
```

What you have learned here about using Spring Java Configuration is at the table when using Spring Boot. Spring Boot applications are all using the Java configuration style, so lets see how you can build Camel unit test 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 very easy as you just add the following dependency to your Maven `pom.xml` file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-boot-starter</artifactId>
    <version>2.17.1</version>
</dependency>
```

And just as when using Spring Java Config your Camel routes can be automatic discovered if you annotate the class with `@Component` as shown:

```
package camelinaction;

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

@Component
public class FileMoveRoute extends RouteBuilder {1
    @Override
    public void configure() throws Exception {
        from("file://target/inbox")
            .to("file://target/outbox");
    }
}
```

① Annotate the class with `@Component` to make Spring discover the class easily

To bootstrap a Spring Boot application becomes even easier than Spring Java Config because of the dependency of `camel-spring-boot-starter` will automatic embed Camel and discover any

Camel routes that has been annotated with `@Component`. To main class to bootstrap Spring Boot is also very easy, there is even no Camel code at all as shown:

```
package camelinaction;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args); ②
    }
}
```

- ① `SpringBootApplication` annotation to mark this class as a Spring Boot application
- ② One line to start the Spring Boot application

All we have to do is to annotate the class with `@SpringBootApplication` ① and then write one line of Java code in the `main` ② method to run the application.

Testing this application is as easy as testing the Spring Java Config application we covered in the previous section. The only difference is that with Spring Boot you use `@SpringApplicationConfiguration` instead of `@ContextConfiguration` to specify which configuration class to use. Listing 9.8 shows the Camel unit test with Spring Boot.

Listing 9.8 Unit testing Camel with Spring Boot

```
import java.io.File;
import org.apache.camel.CamelContext;
import org.apache.camel.Exchange;
import org.apache.camel.ProducerTemplate;
import org.apache.camel.test.spring.CamelSpringBootRunner;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.SpringApplicationConfiguration;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

@RunWith(CamelSpringBootRunner.class)
@SpringApplicationConfiguration(MyApplication.class) ① ②
public class RuntWithFirstTest {

    @Autowired
    private CamelContext context; ③
    @Autowired
    private ProducerTemplate template; ③

    @Test
    public void testMoveFile() throws Exception { ④
        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);
    }
}

```

- ① Use `@RunWith` with the CamelSpringBoot runner
- ② Configure Spring Boot to use `MyApplication` as the configuration class
- ③ Dependency inject `CamelContext` and `ProducerTemplate`
- ④ The unit test method is identical with testing without using `@RunWith`

To run the test as JUnit we use `CamelSpringBootRunner` as `@RunWith` - notice how the runner is named `CamelSpringBootRunner`. Then we refer to the Spring Boot application class which is `MyApplication` ② . The rest is similar to testing with Spring Java Config ④ , where we must dependency inject `CamelContext` and `ProducerTemplate` ③ to make them available for use in our test methods.

We have prepared this example in the source code in the chapter9/spring-boot directory which you can try using the following Maven goal:

```
mvn test -Dtest=FirstTest
```

TIP If you want to learn more about Spring Boot then Manning offers *Spring Boot in Action*, by Craig Walls.

Okay that was a lot of coverage of testing Camel with Spring. Spring is not all there is so lets move on to talk about 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 will learn how to get started writing unit tests for those XML files. Before we get started 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 said OSGi server.

This changed in 2011 when Karl Pauls created PojoSR an OSGi Lite container. PojoSR runs without an OSGi framework and uses flat classloader. However it was good enough where PojoSR will bootstrap a simulated OSGi environment where you co-locate your unit test and Camel dependencies, and it all runs in the same classloader. In other words it runs in the same JVM locally and therefore startup fast, and you can also use the Java debugger. However the flip side is that the more OSGi need you have the bigger chance is that PojoSR has reached its limits. When have such needs you can turn to using an alternative testing framework called Pax-Exam or Arquillian.

We will towards the end of this chapter take a look at those. PojoSR has since become part of Apache Felix as the Felix Connector project. So what you learn in this section is running using Felix Connector.

To get started with 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>2.17.1</version>
</dependency>
```

To keep this example basic we'll use the following OSGi Blueprint XML file with the file copier Camel route as shown in listing 9.9.

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"
    xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
    xmlns:camel="http://camel.apache.org/schema/blueprint"
    xsi:schemalocation="
        http://www.osgi.org/xmlns/blueprint/v1.0.0
        https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

    <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 `CamelTestBlueprintSupport` and write the unit test almost like what we have seen in the previous section. Listing 9.10 shows the code.

Listing 9.10 OSGi Blueprint based unit test

```
import java.io.File;
import org.apache.camel.Exchange;
import org.apache.camel.test.blueprint.CamelBlueprintTestSupport;
import org.junit.Test;

public class BlueprintFirstTest extends CamelBlueprintTestSupport { ❶

    @Override
    protected String getBlueprintDescriptor() { ❷
        return "camelinaction/firststep.xml";
    }

    @Test
    public void testMoveFile() throws Exception { ❸
    }
}
```

```

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);
}
}

```

- ① Test class must extend CamelBlueprintTestSupport
- ② Load the OSGi Blueprint XML file
- ③ Test method is similar to previous examples

As you can see the `BlueprintFirstTest` class extends `CamelBlueprintTestSupport` which is required. Then we override the method `getBlueprintDescription` where we return the location in the classpath of the OSGi Blueprint XML file ② (you can specify multiple files by separating the locations using comma). The rest of the test method is similar to previous examples ③.

This example is located in the `chapter9/blueprint-xml` directory which you can try using the following Maven goal:

```
mvn test -Dtest=BlueprintFirstTest
```

Felix Connect limitations

It's important to know that Felix Connect (aka PojoSR) has its limits how far down the OSGi rabbit hole you can go. Therefore try to write simple unit tests that run with `camel-test-blueprint`, and integration tests than run in the OSGi server using Pax Exam or Arquillian. We'll cover using Pax Exam in section 9.6.2.

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 word.

MOCKING OSGI SERVICES

If you are in one of these teams, how you can unit test your Camel applications where some of these shared services are meant to be part of your Camel application. What you can do with `camel-test-blueprint` is to mock these services and register them manually in the OSGi service registry.

Suppose your Camel Blueprint route depends on an OSGi service as shown below:

```
<reference id="auditService" interface="camelinaction.AuditService"/> ①
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

```

<route>
    <from uri="file://target/inbox"/>
    <bean ref="auditService"/>
    <to uri="file://target/outbox"/>
</route>
</camelContext>

```

- ① Reference to existing OSGi service
- ② Calling the OSGi service from Camel route

The `<reference>` ① is a reference to an existing OSGi service in the OSGi Service Registry which is identified only by the interface which 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`, then the test will fail with an OSGi Blueprint error stating a timeout error waiting for the service to be available.

Yes this is a very 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 to 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:

```

public class MockAuditService implements AuditService {
    public void audit(Exchange exchange) {
        exchange.getContext().createProducerTemplate()
            .send("mock:audit", exchange);
    }
}

```

Then you can register the `MockAuditService` as the fake service in the OSGi Service Registry as part of your unit test using the `addServicesOnStartup` method as shown:

```

protected void addServicesOnStartup(Map<String,
                                    KeyValueHolder<Object, Dictionary>> services) {
    MockAuditService mock = new MockAuditService()
        services.put(AuditService.class.getName(), asService(mock, null)); ①
} ②

```

- ① Create mock of the `AuditService`
- ② Register the mock into the OSGi ServiceRegistry

You can find this example with the source code in the `chapter9/blueprint-xml-service` directory which you can try using the following Maven goal:

```
mvn test
```

We will now move on to use Camel with CDI and demonstrate how to get started testing Camel running in a CDI container such as JBoss Weld and then followed by using a JEE application server such as WildFly.

9.2.5 Camel testing with CDI

Camel has great integration with CDI that allows you to build Camel routes using Java code. To get started with 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>2.17.1</version>
</dependency>
```

We will as usual use a basic Camel example to show you the ropes to get onboard testing with CDI. The Camel route using for testing is a file copier example as shown below:

```
package camelinaction;

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.cdi.ContextName;

@ContextName("helloCamel") ①
public class FileMoveRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("file:target/inbox")
            .to("file:target/outbox");
    }
}
```

① Configure the name of Camel to be helloCamel

The class is annotated with `@ContextName` that allows us to configure the name of the CamelContext. The annotation is optional but it can help readers of the code to better understand that this class will be auto discovered when the CDI application is starting up and the class is part of Camel.

To unit test this Camel route you do not extend a base class but use `CamelCdiRunner` JUnit runner as shown in listing 9.11.

Listing 9.11 Unit testing Camel CDI application

```
package camelinaction;

import java.io.File;
import javax.inject.Inject;
import org.apache.camel.CamelContext;
import org.apache.camel.Exchange;
import org.apache.camel.ProducerTemplate;
import org.apache.camel.cdi.Uri;
import org.apache.camel.test.cdi.CamelCdiRunner;
import org.junit.Test;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
```

```

@RunWith(CamelCdiRunner.class) ①
public class FirstTest {

    @Inject
    private CamelContext context; ②

    @Inject @Uri("file:target/inbox")
    private ProducerTemplate template; ②

    @Test
    public void testMoveFile() throws Exception { ③
        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);
    }
}

```

- ① Use `@RunWith` with the Camel CDI runner
- ② Dependency inject `CamelContext` and `ProducerTemplate`
- ③ The unit test method is identical with previous unit tests covered previously

To run the test as JUnit we use `CamelCdiRunner` as `@RunWith` ① . Because the `FirstTest` class does not extend a base class we need to dependency inject the resources we used during testing such as `CamelContext` and `ProducerTemplate` ② . The test method is implemented similar to what's been covered before in this section ③ .

This example is provided with the source code in the chapter9/cdi directory which you can try using the following Maven goal:

```
mvn test -Dtest=FirstTest
```

The example can also run standalone as it includes a main class as entry which you can run from Maven with:

```
mvn compile exec:java
```

You can also run Camel application in JEE application servers such as WildFly but how do you do testing with Camel and WildFly?

9.2.6 Camel testing with WildFly

There are several ways of testing Camel with WildFly. The most simplest form would be a regular unit test that runs standalone outside WildFly. In other words an unit test that would be much the same we have already covered by:

- 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 us to do system tests with *the real thing* (where we test using the actual used application server).

We will build a simple example that uses JEE technology together with Camel running in 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 an WAR deployment running inside the WildFly server.

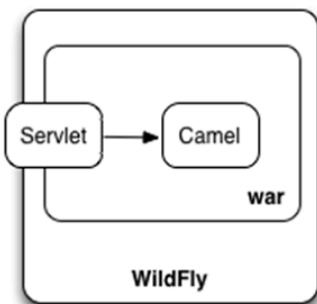


Figure 9.3 A WAR deployment using servlet and Camel running in WildFly application server.

The servlet is implemented as shown in listing 9.12.

Listing 9.12 A hello Servlet that calls a Camel route

```

package camelinaction;

import java.io.IOException;
import javax.inject.Inject;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.camel.ProducerTemplate;

@WebServlet(name = "HelloServlet", urlPatterns = {"/*"}, loadOnStartup = 1) ①
public class HelloServlet extends HttpServlet {

    @Inject
    private ProducerTemplate producer; ②

    protected void doGet(HttpServletRequest req,
                         HttpServletResponse res) throws IOException {
        String name = req.getParameter("name");

        String s = producer.requestBody("direct:hello", name, String.class); ③
    }
}
  
```

```

        res.getOutputStream().print(s);
    }
}

① @WebServlet to make this class act as a Servlet
② Dependency inject a Camel ProducerTemplate using CDI injection
③ Call the Camel route to retrieve an output to use as response

```

The servlet is implemented in the `HelloServlet` class and by annotating the class with `@WebServlet` we do not have to register the servlet using the `WEB-INF/web.xml` file. The servlet integrates with Camel by having a `ProducerTemplate` injected using CDI 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 very simple route that does a message transformation as shown:

```

@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 which can be deployed to any JEE web server such as Apache Tomcat or WildFly. In this example we are using WildFly and will now show how you can test your deployments running inside WildFly.

To be able to test this we use a testing framework called Arquillian that allows to write unit tests that are able to deploy your application and unit 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 life cycle management. Instead of managing a runtime in your test, Arquillian brings your test to the runtime.

<http://arquillian.org/modules/core-platform/>

TESTING WILDFLY USING ARQUILLIAN

To use Arquillian to perform container tests using WildFly you need to add the following dependencies to your Maven `pom.xml` file:

```

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.shrinkwrap.resolver</groupId>
            <artifactId>shrinkwrap-resolver-bom</artifactId>
            <version>2.2.2</version>

```

①

```

<scope>import</scope>
<type>pom</type>
</dependency>
<dependency>
  <groupId>org.jboss.arquillian</groupId>
  <artifactId>arquillian-bom</artifactId>
  <version>1.1.11.Final</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
</dependencies>
</dependencyManagement>

<dependency>
  <groupId>org.jboss.arquillian.junit</groupId>
  <artifactId>arquillian-junit-container</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.shrinkwrap.resolver</groupId>
  <artifactId>shrinkwrap-resolver-impl-maven</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.wildfly.arquillian</groupId>
  <artifactId>wildfly-arquillian-container-managed</artifactId>
  <version>1.0.2.Final</version>
  <scope>test</scope>
</dependency>

```

- ➊ Import ShrinkWrap Maven Resolver BOM
- ➋ Import Arquillian BOM
- ➌ Arquillian module for JUnit tests
- ➍ ShrinkWrap Maven Resolver Implementation
- ➎ Arquillian WildFly module

Arquillian allows to use ShrinkWrap ➊ ➍ to resolve the dependencies to include in the deployment unit(s) used during testing. Arquillian provides a JUnit runner ➌ which we use in the unit test shown in listing 9.13, that runs in WildFly ➎.

Listing 9.13 System test that tests our Servlet and Camel application running in WildFly

```

package camelinaction;

import java.io.File;
import javax.inject.Inject;
import org.apache.camel.CamelContext;
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.jboss.shrinkwrap.api.spec.WebArchive;
import org.jboss.shrinkwrap.resolver.api.maven.Maven;
import org.junit.Test;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertEquals;

```

```

@RunWith(Arquillian.class)
public class FirstWildFlyTest { 1

    @Inject
    CamelContext camelContext; 2

    @Deployment
    public static WebArchive createDeployment() {
        File[] files = Maven.resolver()
            .loadPomFromFile("pom.xml")
            .importRuntimeDependencies()
            .resolve().withTransitivity().asFile(); 3

        WebArchive archive = ShrinkWrap.create(WebArchive.class,
            "mycamel-wildfly.war");
        archive.addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml");
        archive.addPackages(true, "camelinaction"); 4
        archive.addAsLibraries(files); 5
        return archive;
    } 6

    @Test
    public void testHello() throws Exception { 7
        String out = camelContext.createProducerTemplate()
            .requestBody("direct:hello", "Donald", String.class);
        assertEquals("Hello Donald", out);
    }
}

```

- 1 JUnit runner to run using Arquillian
- 2 Dependency inject CamelContext
- 3 ShrinkWrap to resolve all the runtime dependencies from the Maven pom.xml file
- 4 Create WebArchive deployment using ShrinkWrap
- 5 Include the projects classes
- 6 Add all the Maven dependencies as libraries in the deployment unit
- 7 The test method that calls the Camel route

The test class `FirstWildFlyTest` uses the Arquillian JUnit runner ① that lets Arquillian in charge of the tests. Then we use CDI to inject `CamelContext` into the test class ②. To include all needed dependencies in the deployment unit we use ShrinkWrap Maven resolver ③ that includes all the runtime dependencies and their transitive dependencies. The deployment unit is then created ④ using ShrinkWrap as a WAR deployment with the name `mycamel-wildfly.war`. Its important you use the correct name of the WAR file as configured in the Maven `pom.xml` file:

```
<finalName>mycamel-wildfly</finalName>
```

The method `addPackages` ⑤ includes all the classes from the root package `camelinaction` and all sub packages. This ensures all the classes of this project is included in the deployment unit. The method `addAsLibraries` ⑥ adds all the dependencies ③ as JARs to the deployment unit. The actual unit test method is a simple test that calls the Camel route ⑦.

We could have chosen to perform a HTTP call to call the servlet instead, however then we need to add a HTTP library. We cheat a bit and just call the Camel route.

Okay that was a mouthful to take in are we ready for testing? No we still have two tasks to do. Setup Arquillian and install WildFly.

You configure Arquillian in the arquillian.xml file which you put in the src/test/resources directory and its content is as shown below:

```
<?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> ①
  </engine>
  <container qualifier="managed" default="true"> ②
    <configuration>
      <property name="jbossHome">${env.JBOSS_HOME}</property> ③
      <property name="serverConfig">standalone.xml</property>
      <property name="allowConnectingToRunningServer">true</property> ④
    </configuration>
  </container>
</arquillian>
```

- ① To use the target directory during deployment and testing
- ② Use a managed WildFly container
- ③ The location of where JBoss WildFly is installed
- ④ To use the standalone.xml configuration with WildFly

The arquillian.xml configuration file fairly easy to comprehend. At first we configure to use the target directory ① as work directory so its easy to cleanup using Maven clean goal. We use Arquillian container testing in managed mode ②. This means an existing installation of WildFly is used, so we need to configure the directory where WildFly is installed. The recommended way is to JBOSS_HOME environment variable for that ③. Then we 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 would be similar principle when 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 setup the JBOSS_HOME environment variable.

The author of this chapter has installed WildFly 10 in the /opt directory and has setup the JBOSS_HOME environment as:

```
export JBOSS_HOME=/opt/wildfly-10.0.0.Final
```

After this is done you are ready for running the system tests, which is as simple as running Maven test. We have includes this example in the source code in the chapter9/wildfly directory which you can try using:

```
mvn test -Dtest=FirstWildFlyTest
```

We took our time to walk you through how to setup a project for doing system tests with Arquillian that runs in WildFly. What you have learned here will come back handy in section 9.6 where we cover more about system tests using different test libraries such as Arquillian, Citrus, and Pax-Exam.

You have now seen the Camel Test Kit and learned to use its JUnit extension to write your first unit tests using a number of different 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 leverages many protocols has always been challenging.

This is why mocks were invented. By 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 would be a good time to take a little break because we are changing the scene from running Camel tests on various runtimes to cover all about the Camel mock component, that are very useful to use 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 as 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 only have access to the component in 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 would 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 would be to test using the real component, which is usually much harder. You may already have used mocking before; there are many frameworks out there that blend in well with testing frameworks like JUnit.

Camel takes testing very seriously, and the Mock component was included in the first release of Camel. The fact that it resides in camel-core JAR indicates its importance—the Mock component is used rigorously in unit testing Camel itself.

In this section, we'll look at how to use the Mock component in unit tests and how to add mocking to existing unit tests. Then we'll spend some time on how you can use 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

The three basic steps of testing are illustrated in figure 9.4.

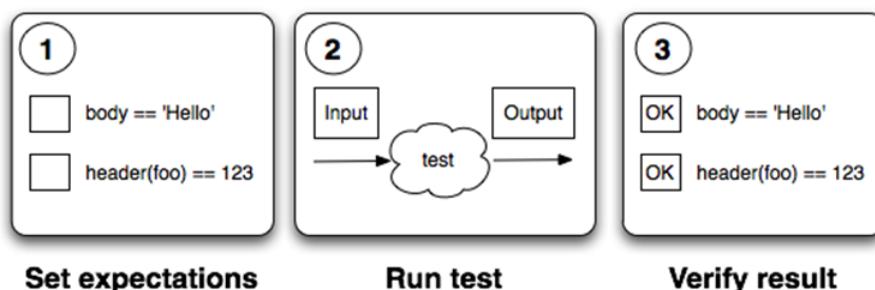


Figure 9.4 Three steps for testing: set expectations, run the test, and verify the result

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:

- That the correct number of messages are received on each endpoint
- That the messages arrive in the correct order
- That the correct payloads are received
- That the test ran within the expected time period

Mock components allow you to configure coarse- and fine-grained expectations and to 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 we look at how to use the Mock component, we'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 9.2 lists the most commonly used methods on the mock endpoint.

Table 9.2 Commonly used methods in the MockEndpoint class

| Method | Description |
|--|---|
| <code>expectedMessageCount (int count)</code> | Specifies the expected number of messages arriving at the endpoint |
| <code>expectedMinimumMessageCount (int count)</code> | Specifies the expected minimum number of messages arriving on 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 listing 9.14.

Listing 9.14 Using MockEndpoint in unit testing

```
package camelinaction;

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.mock.MockEndpoint;
import org.apache.camel.test.junit4.CamelTestSupport;

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

    template.sendBody("jms:topic:quote", "Camel rocks");

    quote.assertIsSatisfied(); 2
}

```

- ① Expects one message
- 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 or not 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 9.13 to what you saw in figure 9.4: you set expectations, ran the test, and verified 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(long timeInMillis)` method if you have unit tests that run for a long time. However it's easier to specify the timeout 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 simulating JMS using the Stub component (We'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 our example we have the endpoint `jms:topic:quote` which can be stubbed as `stub:jms:topic:quote`. The route will be as simple as:

```
from("stub:jms:topic:quote").to("mock:quote");
```

The producer can then send a message to the queue using

```
template.sendBody("stub:jms:topic:quote", "Camel rocks");
```

We have provided this example in the source code in chapter9/mock directory which you can try using the following Maven goals:

```
mvn test -Dtest=FirstMockTest
mvn test -Dtest=SpringFirstMockTest
```

You may have noticed in listing 9.14 that the expectation was coarse-grained in the sense that you just expected one message to arrive. You did not specify anything about the message's content or other characteristics, so you don't know whether the message that arrived was the same "Camel rocks" message that was sent. The next section covers how to test this.

9.3.3 Verifying that the correct message arrived

The `expectedMessageCount` method can only be used to verify that a certain number of messages arrived. It doesn't dictate anything about the content of the message. Let's improve the unit test in listing 9.14 so that it expects the message being sent to match the message that arrives at the mock endpoint.

You can do this using the `expectedBodiesReceived` method, as follows:

```
@Test
public void testQuote() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:quote");
    mock.expectedBodiesReceived("Camel rocks");

    template.sendBody("jms:topic:quote", "Camel rocks");

    mock.assertIsSatisfied();
}
```

This is intuitive and easy to understand, but the method states bodies in 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("jms:topic:quote", "Camel rocks");
    template.sendBody("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.

In cases where the order doesn't matter, you can use the `expectedBodiesReceived-InAnyOrder` 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 bodies you pass in as an argument will be very 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 chapter9/mock directory which you can try using the following Maven goals:

```
mvn test -Dtest=FirstMockTest
mvn test -Dtest=SpringFirstMockTest
```

The Mock component has many other features we need to cover, so let's continue and see how you can 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. One way of doing this is shown in listing 9.15.

Listing 9.15 Using expressions with MockEndpoint to set expectations

```
@Test
public void testIsCamelMessage() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:quote");
    mock.expectedMessageCount(2); ①

    template.sendBody("jms:topic:quote", "Hello Camel");
    template.sendBody("jms:topic:quote", "Camel rocks");

    assertMockEndpointsSatisfied(); ②

    List<Exchange> list = mock.getReceivedExchanges();
    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"));
}
```

- ① Expects 2 messages
- ② Verifies 2 messages received
- ③ Verifies "Camel" is in received messages

First you set up your expectation that the `mock:quote` endpoint will receive two messages ①. You then send in two messages to the JMS topic to start the test. Then 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, of course 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 9.3 lists some additional `MockEndpoint` methods that let you use expressions to set expectations.

Table 9.3 Expression-based methods commonly used on `MockEndpoint`

| Method | Description |
|---|---|
| <code>message(int index)</code> | Defines an expectation on the <code>'n</code> th message received |
| <code>allMessages()</code> | Defines an expectation on all messages received |
| <code>expectsAscending(Expression expression)</code> | Expects messages to arrive in ascending order |
| <code>expectsDescending(Expression expression)</code> | Expects messages to arrive in descending order |
| <code>expectsDuplicates(Expression expression)</code> | Expects duplicate messages |
| <code>expectsNoDuplicates(Expression expression)</code> | Expects no duplicate messages |
| <code>expects(Runnable runnable)</code> | 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("jms:topic:quote", "Hello Camel");
    template.sendBody("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 only seen expectations based on 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 we used in the preceding couple of code snippets. They're builder methods used to create predicates for expectations. Table 9.4 lists all the builder methods available.

Table 9.4 Builder methods for creating predicates to be used as expectations

| Method | Description |
|---|---|
| <code>contains(Object value)</code> | Sets an expectation that the message body contains the given value |
| <code>isInstanceOf(Class type)</code> | Sets an expectation that the message body is an instance of the given type |
| <code>startsWith(Object value)</code> | Sets an expectation that the message body starts with the given value |
| <code>endsWith(Object value)</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(Object value)</code> | Sets an expectation that the message body is equal to the given value |
| <code>isNotEqualTo(Object value)</code> | Sets an expectation that the message body isn't equal to the given value |
| <code>isGreaterThan(Object value)</code> | Sets an expectation that the message body is greater than the given value |
| <code>isGreaterThanOrEqualTo(Object value)</code> | Sets an expectation that the message body is greater than or equal to the given value |
| <code>isLessThan(Object value)</code> | Sets an expectation that the message body is less than the given value |
| <code>isLessThanOrEqualTo(Object value)</code> | Sets an expectation that the message body is less than or equal to the given value |
| <code>isNull(Object value)</code> | Sets an expectation that the message body is <code>null</code> |
| <code>isNotNull(Object value)</code> | Sets an expectation that the message body isn't <code>null</code> |
| <code>matches(Expression expression)</code> | 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, Mvel etc. |
| <code>regex(String pattern)</code> | Sets an expectation that the message body matches the given regular expression |

At first it may seem odd that the methods in table 9.4 often use `Object` as the parameter type—why not a specialized type such as `String`? This is 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(".");
```

You have learned a lot about how to set expectations, including fine-grained ones using the builder methods listed in table 9.4. Now it’s time to move on and test the ordering of the messages received.

9.3.5 Testing the ordering of messages

Suppose you need to test that messages arrive in sequence-number order. For example, messages arriving in the order 1, 2, 3 are accepted, whereas the order 1, 3, 2 is invalid and the test should fail.

The Mock component provides features to test ascending and descending orders. For example, you can use the `expectsAscending` method like this:

```
mock.expectsAscending(header("Counter"));
```

The preceding expectation will test that the received messages are in ascending order, judged by the `Counter` value in the message header, but it doesn’t dictate what the starting value must be. If the first message that arrives has a value of 5, the expectation tests whether or not the next message has a value greater than 5, and so on.

What if you must test that the first message has a value of 1? In that case, you can add another expectation that tests the first message, using `message(0)`, as follows:

```
mock.message(0).header("Counter").isEqualTo(1);
mock.expectsAscending(header("Counter"));
```

Together these expectations test that messages arrive in the order 1, 2, 3, ..., but orders such as 1, 2, 4, 5, 6, 8, 10, ... also pass the test. That’s because the `expectsAscending` and `expectsDescending` methods don’t detect whether there are *gaps* between messages. These methods use generic comparison functions that work on any types, not only numbers.

To detect gaps in the sequence, you need to use a custom expression that implements gap-detection logic.

USING A CUSTOM EXPRESSION

When the provided expressions and predicates don't cut it, you can use a custom expression. By using a custom expression, you have the full power of Java code at your fingertips to implement your assertions.

Let's look at the problem of gap detection. Camel doesn't provide any expressions for that, so you must do it with a custom expression. Listing 9.16 shows how this can be done.

Listing 9.16 Using a custom expression to detect gaps in message ordering

```
@Test
public void testGap() throws Exception {
    final MockEndpoint mock = getMockEndpoint("mock:quote");
    mock.expectedMessageCount(3);
    mock.expects(new Runnable() { ①
        public void run() {
            int last = 0;
            for (Exchange exchange : mock.getExchanges()) {
                int current = exchange.getIn()
                    .getHeader("Counter", Integer.class);
                if (current <= last) {
                    fail("Counter is not greater than last counter");
                } else if (current - last != 1) {
                    fail("Gap detected: last: " + last + " current: " + current);
                }
                last = current;
            }
        }
    });
    template.sendBodyAndHeader("jms:topic:quote", "A", "Counter", 1);
    template.sendBodyAndHeader("jms:topic:quote", "B", "Counter", 2);
    template.sendBodyAndHeader("jms:topic:quote", "C", "Counter", 4);

    mock.assertIsNotSatisfied();
}
```

① Custom expression to detect gaps

To set up a custom expression, you use the `expects` method, which allows you to provide your own logic as a `Runnable` ①. In the `Runnable`, you can loop through the exchanges received and extract the current counter header. Then you can verify whether all the counters are incremented by one and don't have any gaps. You can use the JUnit `fail` method to fail when a gap is detected.

TIP If you develop using Java 8 style you can a lambda to represent the inlined `Runnable` in listing 9.16.

To test whether this works, you send in three messages, each of which contains a Counter. Notice that there is a gap in the sequence: 1, 2, 4. You expect this unit test to fail, so you instruct the mock to *not* be satisfied using the `assertIsNotSatisfied` method.

Next, you test a positive situation where no gaps exist. To do so, you use the `assertIsSatisfied` method and send in three messages in sequence, as follows:

```
template.sendBodyAndHeader("seda:topic:quote", "A", "Counter", 1);
template.sendBodyAndHeader("seda:topic:quote", "B", "Counter", 2);
template.sendBodyAndHeader("seda:topic:quote", "C", "Counter", 3);
mock.assertIsSatisfied();
```

That's all there is to developing and using a custom expression.

You can try this example from the chapter9/mock directory by running the following Maven goal:

```
mvn test -Dtest=GapTest
```

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.

9.3.6 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 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 have been 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 actual order status. You have been asked to simulate this server by replying with a canned response.

Camel provides the two methods listed in table 9.5 to help simulate a real component.

Table 9.5 Methods to control responses when simulating a real component

| Method | Description |
|--|--|
| whenAnyExchangeReceived (Processor processor) | Uses a custom processor to set a canned reply |
| whenExchangeReceived (int index, Processor processor) | Uses a custom processor to set a canned reply when the 'th message is received |

You can simulate a real endpoint by mocking it with the Mock component and controlling the reply using the methods in table 9.5. To do this, you need to replace the actual 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 leverages the preceding route follows.

Listing 9.17 Simulating a real component by using a mock endpoint

```
public class MirandaTest extends CamelTestSupport {

    @Override
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            public void configure() throws Exception {
                from("jetty:http://localhost:9080/service/order")
                    .process(new OrderQueryProcessor())
                    .to("mock:miranda")
                    .process(new OrderResponseProcessor());
            }
        };
    }

    @Test
    public void testMiranda() throws Exception {
        MockEndpoint mock = getMockEndpoint("mock:miranda");
        mock.expectedBodiesReceived("ID=123");
        mock.whenAnyExchangeReceived(new Processor() {
            public void process(Exchange exchange) throws Exception {
                exchange.getIn().setBody("ID=123,STATUS=IN PROGRESS");
            }
        });
    }

    String url = "http://localhost:9080/service/order?id=123";
    String out = template.requestBody(url, null, String.class);
    assertEquals("IN PROGRESS", out);
}
```

①

②

```

        assertMockEndpointsSatisfied();
    }

    private class OrderQueryProcessor implements Processor { ③
        public void process(Exchange exchange) throws Exception {
            String id = exchange.getIn().getHeader("id", String.class);
            exchange.getIn().setBody("ID=" + id);
        }
    }

    private class OrderResponseProcessor implements Processor { ④
        public void process(Exchange exchange) throws Exception {
            String body = exchange.getIn().getBody(String.class);
            String reply = ObjectHelper.after(body, "STATUS=");
            exchange.getIn().setBody(reply);
        }
    }
}

```

- 1 Returns canned response
- 2 Verifies expected reply
- 3 Transforms to format understood by Miranda
- 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 ①, 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" using the regular JUnit `assertEquals` method ②. You use two processors (③ and ④) to transform the data to and from the format that the Miranda server understands.

You can find the code for this example in the `chapter9/miranda` folder of the book's source code, which you can try using the following Maven goal:

```
mvn test -Dtest=MirandaTest
```

You've now learned all about the Camel Test Kit and how to use it for unit testing with Camel. We 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 is a more cunning way to simulate a real component than by using a mock, and there is. We're going to look at how to simulate 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 premise and within 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 will have a long walk and have to go all Chuck Norris to enter the guarded data centers from Amazon, Google or whom are your cloud providers. That's sadly not going to happen in our lifetime, and we can only dream about becoming Neo in Matrix whom is capable of learning ninja and kung-fu skills in a matter of minutes.

So back to our earthly lives we would have to come up with something you can do from your computer. Instead we'll look at how to simulate errors in unit tests using the three different techniques listed in table 9.6.

Table 9.6 Three techniques for simulating errors

| Technique | Description |
|-------------|---|
| Processor | <p>Using processors is easy, and they give you full control, as a developer.</p> <p>This technique is covered in section 9.4.1.</p> |
| Mock | <p>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.</p> <p>This technique is covered in section 9.4.2.</p> |
| Interceptor | <p>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.</p> <p>We'll cover interceptors in section 9.4.3.</p> |

The following three sections covers these three techniques.

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 the forthcoming error handler chapter—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=1h")
.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 will 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 actual components used, you can just mock the HTTP and FTP 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:start")
    .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 an actual 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 connection 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
```

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 9.3.6 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, we 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. Listing 9.18 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/errors directory you can try 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 listing 9.16 that should, among other things, test what happens when communication with the remote HTTP server fails. How can you do that? This is 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 we 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 upon it. Figure 9.5 illustrates where the interception takes place in a route.

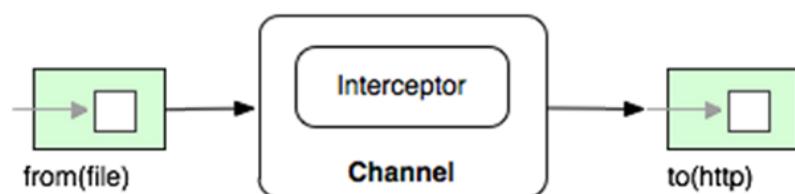


Figure 9.5 The channel acts as a controller, and it's where message are intercepted during routing.

Figure 9.5 shows a low-level view of a Camel route, where 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 Camel routing engine, handling such things as routing the message to the next designated target, error handling, interception, tracing messages and gathering metrics.

The three types of interceptors that Camel provides out of the box are listed in table 9.7.

Table 9.7 The three flavor 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 on a particular endpoint. This interceptor is only invoked once. |
| interceptSendToEndpoint | Intercepts messages that are about to be sent to a particular endpoint. This interceptor is only invoked 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.

You can find an example from the source code in `chapter9/errors` you can try using the following Maven goal:

```
mvn test -Dtest=SimulateErrorUsingMockTest
```

TIP The last two interceptors in table 9.7 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 is available during unit testing, and it 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 you can use `adviceWith` in a unit test method:

```
@Test
public void testSimulateErrorUsingInterceptors() throws Exception {
    RouteDefinition route = context.getRouteDefinitions().get(0);
    route.adviceWith(context, new RouteBuilder() {1
        public void configure() throws Exception {2
            interceptSendToEndpoint("http://*")
                .skipSendToOriginalEndpoint()
                .process(new SimulateHttpErrorProcessor());
        }
    });
    ..
}
```

- ① Select the first route to be advised
- ② Uses `adviceWith` to add interceptor to route

The key issue when using `adviceWith` is to know which route to use. Because you only have 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've got the route, it's just a matter of using the `adviceWith` method ② , which leverages a `RouteBuilder`—this means that 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 it 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.

In this example we only scratched the surface of advice with. Lets 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 where the routes are *advice* before testing. Some of the things you can do by advising are:

- Intercept sending to endpoints
- Replacing 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 following additional fluent builder methods listed in table 9.8.

Table 9.8 Additional advice with methods from AdviceWithRouteBuilder.

| Method | Description |
|-----------------------------------|--|
| mockEndpoints | Mock all endpoints in the route. |
| mockEndpoints(patterns...) | Mock 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. |
| mockEndpointsAndSkip(patterns...) | Mock all endpoints and skip sending to the actual endpoint in the route that matches the pattern(s). You can use wildcards and regular expressions in the given pattern to match multiple endpoints. |
| replaceFromWith(uri) | To easily replace the incoming endpoint in the route |
| weaveById(pattern) | Is used to manipulate the route at the node id's that matches the pattern. Using weave is covered in section 9.4.6. |
| weaveByToString(pattern) | Is used to manipulate the route at the node <code>toString</code> representation that matches the pattern. Using weave is covered in section 9.4.6. |
| weaveByType(pattern) | Is used to manipulate the route at the node type's that matches the pattern. Using weave is covered in section 9.4.6. |
| weaveAddFirst | To easily weave in new nodes at the beginning of the route. Using weave is covered in section 9.4.6. |
| weaveAddLast | To easily weave in new nodes at the end of the route. Using weave is covered in section 9.4.6. |

We will in the following cover the first four methods from table 9.8, and then all the weave methods are covered in the next section.

TELL CAMEL THAT YOU ARE ADVISING ROUTES

But before we jump into the pool we need to learn a rule from the life guard. When `adviceWith` is being used Camel will re-start the advised routes. The reason is that `adviceWith` essentially manipulates the layout of the route, and therefore Camel has to:

- 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. This happens very quickly that a route is just started up, and then immediately being stopped and removed. This is an undesired behavior and we want to avoid re-starts 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 start the routes during startup.
2. Advice the routes in your unit test methods
3. Start Camel after you have advised the routes

The first step is done by overriding the `isUseAdviceWith` method from `CamelTestSupport` and return true as shown:

```
public class MyAdviceWithTest extends CamelTestSupport {

    @Override
    public boolean isUseAdviceWith() {
        return true;
    }
    ...
}
```

- ① Tell Camel that we intend to use adviceWith in this unit test**

Then you advise the routes followed by starting Camel as shown:

```
@Test
public void testMockEndpoints() throws Exception {
    RouteDefinition route = context.getRouteDefinition("quotes");
    route.adviceWith(context, new AdviceWithRouteBuilder() {           ①
        public void configure() throws Exception {
            mockEndpoints();                                         ②
        }
    });
    context.start();                                                 ③
    ...
}
```

- ① Advising the route**
② Auto mocking all the endpoints
③ Start Camel after the advice is done

To advise a route we need to obtain the route using its route id. Then we can use the `AdviceWithRouteBuilder` ① that allows us to use the advice methods from table 9.8, such as auto mocking all the Camel endpoints. After we are done with the advice we need to start Camel that would then start the routes after they have been advised, and thus no re-start of the routes is performed anymore.

This example is provided as part of the source code in the `chapter9/advicewith` directory which you can try using the following Maven goal:

```
mvn test -Dtest=AdviceWithMockEndpointsTest
```

What was the important message from the life guard? To tell Camel that you are using advice with.

REPLACE ROUTE ENDPOINTS

You may have build Camel routes that starts from endpoints that consumes 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. Listing 9.19 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
            }
        });
        context.start();                                         ⑤

        getMockEndpoint("mock:seda:camel")                     ⑥
            .expectedBodiesReceived("Camel rocks");
        getMockEndpoint("mock:seda:other")                     ⑥
            .expectedBodiesReceived("Bad donkey");

        template.sendBody("direct:hitme", "Camel rocks");      ⑦
        template.sendBody("direct:hitme", "Bad donkey");       ⑦

        assertMockEndpointsSatisfied();                         ⑧
    }

    @Override
    protected RoutesBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            public void configure() throws Exception {
                from("aws-sqs:quotes").routeId("quotes")          ⑨
                    .choice()
                        .when(simple("${body} contains 'Camel'")).to("seda:camel")
                        .otherwise().to("seda:other");
            }
        };
    }
}
```

- ① This test uses adviceWith
- ② Get the route to be advised
- ③ Replace the route input endpoint with a direct endpoint
- ④ Mock all SEDA endpoints

- 5 Start Camel after we have done the advise
- 6 Set expectations on the mock endpoints
- 7 Send in test messages to the direct endpoint
- 8 Assert the test passed by asserting the mocks
- 9 The route originally consumes from Amazon SQS queue system

The `ReplaceFromTest` class from listing 9.18 is using `adviceWith` so we need to override the `isUseAdviceWithMethod` and return true ① . Then we select the route to be advised ② using the `AdviceWithRouteBuilder`. The original input route is using the AWS-SQS component ⑨ which we replace to use direct component instead ③ . The route will send messages to other endpoints such as seda. To make testing easier we auto mock all seda endpoints ④ . After the advise is done we need to start Camel ⑤ . Because all the seda endpoints was auto mocked, we can use `mock:` as prefix to obtain their mocked endpoint. This allows us to set expectation on those mock endpoints ⑥ . Then a couple of test messages is send into route using the replaced endpoint which is `direct:hitme` ⑦ . Finally we check if the test passed by asserting the mocks ⑧ .

You can try this example with the source code from chapter9/advicewith directory using the following Maven goal:

```
mvn test -Dtest=ReplaceFromTest
```

Replacing the input endpoint in Camel routes using advice with is just the beginning of the route manipulation capabilities that are available. In the following section we will cover how you can rip and tear your routes as you please.

9.4.6 Using weave with advice with to amend routes

When testing your Camel routes you can use advice with to weave the routes before testing.

Naming in IT

Some IT professionals says that naming is hard, and so were they right in terms of naming advice with and weave. Reflecting back on this then we should have chosen amend as the word instead of weave. As what you do is amending the routes before testing.

You can use this to remove or replace parts of the routs that may not be relevant for a particular unit test where you test other aspects of the routes. Another use-case is to replace parts of routes that coupling to systems that are not available for testing, or may otherwise slow testing down.

In this section we will try some of weave methods from the `AdviceWithRouteBuilder` that was listed in table 9.8.

We will start with a simple route that does 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 would require connection to an external system that you do not want to perform in an unit test. What you can do is to use `weaveById` to select it, and then replace it with another kind of transformation as shown below:

```
public void testWeaveById() throws Exception {
    RouteDefinition route = context.getRouteDefinition("quotes");
    route.adviceWith(context, new AdviceWithRouteBuilder() {
        public void configure() throws Exception {
            weaveById("transform").replace()
                .transform().simple("${body.toUpperCase()}"); ①  

①
            weaveAddLast().to("mock:result"); ②
②
        }
    });
    context.start(); ③
③
    getMockEndpoint("mock:result").expectedBodiesReceived("HELLO CAMEL"); ④
    template.sendBody("seda:quotes", "Hello Camel");
    assertMockEndpointsSatisfied();
} ④
```

- ① Use weave to select the node to be replaced
- ② Route to a mock endpoint at the end of the route
- ③ Remember to start Camel after advice with
- ④ Expect the message to be in upper case

In the original route the bean call was given the id `transform`, which is the id we let `weaveById` ① use as select criteria. Then we select an *action* which can be either of the following:

- `remove` - Removes the selected node(s)
- `replace` - Replaces the selected node(s) with the following
- `before` - Before each of the selected node the following is added
- `after` - After each of the selected node the following is added

In this example we want to replace the node, so we use `replace` which is using the Java DSL where you can define a "mini" Camel route that is being the replacement. In this example we replaced the bean with a `transform`.

You are not restricted to only do a 1:1 replacement, so you could for example provide before and after logging as shown:

```
weaveById("transform").replace()
    .log("Before transformation")
    .transform().simple("${body.toUpperCase()}")
    .log("Transformation done")
```

To make testing easier we use `weaveAddLast` ② to route to a mock endpoint at the end of the route. Remember to start Camel ③ before we start send messages into the route and assert the test is correct ④.

You can play with this example from the source code in chapter9/advice with by running this Maven goal:

```
mvn test -Dtest=WeaveByIdTest
```

Using `weaveById` is recommended to use if you have assigned id's to the EIPs in your Camel routes. If this is not 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 9.8 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 have prepared a little example for you using the Splitter EIP in the route shown below:

```
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 is using a flexible fluent builder (highlighted in bold) that allows to build common strategies. You can use this builder by static importing the `org.apache.camel.util.toolbox.AggregationStrategies.flexible` method.

Because the route only has one Splitter EIP then we can use `weaveByType` to find this single splitter in the route. Using `weaveByType` requires to pass in the model type of the EIP. The name of the model type are using the naming pattern `nameDefinition`, so the splitter is named `SplitDefinition` as highlighted below:

```
weaveByType(SplitDefinition.class)
    .before()
        .filter(body().contains("Donkey"))
        .transform(simple("${body},Mules cannot do this"));
```

Here we 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 if the message body contains the word `Donkey`. And if so it performs a message transformation by appending a quote to the existing message body.

As usual we have prepared this example of wisdom you can try from the chapter9/advice with directory using the following Maven goal:

```
mvn test -Dtest=WeaveByTypeTest
```

Okay this example only has one splitter. But what if I have more than one splitter or want to weave a frequently used type such as send to?

SELECTING ONE OR MORE USING WEAVE

When using the `weaveBy` methods they select all matches nodes, which can be anything from none, one, two or more nodes. In those situations you may want to narrow down the selection to a specific node. This can be done 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)` - To limit the selection to at most N level deep in the Camel route tree. The first level is starting from number 1. So number 2 is the children of the 1st level nodes.

Given the route shown below, you want to select the highlighted sent 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 only match the first found as shown:

```
weaveByType(ToDefinition.class).selectFirst().replace().to("mock:line");
```

You can try this example from the accompanying source code in the chapter9/advicewith directory using the following Maven goal:

```
mvn test -Dtest=WeaveByTypeSelectFirstTest
```

If the route uses more sent to EIPs then the select criteria changes. You would then need to use `selectByIndex` and count the number of to's from top to bottom, to find what index to use.

That was all folks we wanted to talk about advice with. Its a powerful feature in the right hands, so we recommend you give it a shoot and play with the examples and thinker with the source code to get your fingers dirty.

The two last sections of this chapter covers how to do Camel integration testing. Section 9.5 is focused on integration testing using the Camel test kit where as section 9.6 goes out of town where we look at using three different third party test frameworks with Camel.

Do you need a break? If so its a good time to put on the kettle for a cup of tea or coffee. The last two sections are cozy and 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. For example, integration testing often involves real live components, and substituting mocks isn't an option, as the point of the integration test is to test with live components. In this section, we'll look at how to test such situations without using 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.

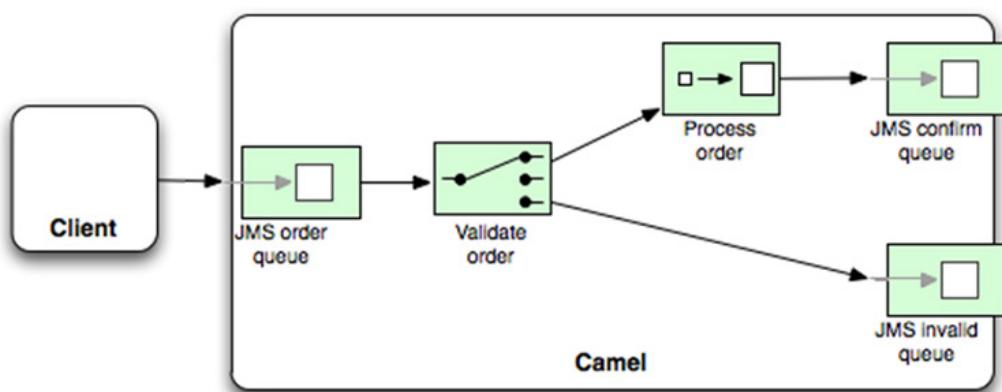


Figure 9.6 The client sends orders to an order queue, which is routed by a Camel application. The order is either accepted and routed to a confirm queue, or it's not accepted and is routed to an invalid queue.

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 Integration testing

Integration testing the scenario outlined in figure 9.6 requires you to use live components, which means 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 if the message arrived as expected

So 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. What you need is 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 the `NotifyBuilder`, which provides such insight. We'll cover the `NotifyBuilder` in more detail in section 9.6.2, but the following code shows how to have `NotifyBuilder` notify 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 the 5-second timeout occurs.

The last task tests whether the `NotifyBuilder`. If it did not match then its because the condition was not meet within the timeout 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. By using the `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 in 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,123,2016-04-20T15:47:58,4444,5555", body);
```

By 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, which you can try using the following Maven goal:

```
mvn test -Dtest=OrderTest
```

We've now covered an example of how to do integration testing without mocks. Along the road, we introduced the `NotifyBuilder`, which has many more nifty features. We'll review it 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 how you can 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 very coarse-grained condition. Suppose you have multiple routes, and another message was processed as well. That would cause the condition to match even if the message you wanted to test was 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 only applies for 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 this using a predicate to indicate when the desired message was completed. This is done using the `whenAnyDoneMatches` method, as shown here in bold:

```
NotifyBuilder notify = new NotifyBuilder(context)
```

```
.from("activemq:queue:order")
    .whenAnyDoneMatches(
        body().isEqualTo("OK,123,2016-04-20'T'15:48:00,2222,3333")
    ).create();
```

In this example we want the predicate to determine if 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 9.9 lists the most commonly used methods.

Table 9.9 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 JMS 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 routId to match multiple routes. |
| <code>filter(predicate)</code> | Filters unwanted messages. |
| <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 matched, or <code>false</code> if a timeout occurred. |

The `NotifyBuilder` has over 30 methods, and we've only listed the most commonly used ones in table 9.9. Consult the online Camel documentation to see all the supported methods: <http://camel.apache.org/notifybuilder.html>.

NOTE The `NotifyBuilder` works in principle by adding an `EventNotifier` to the given `CamelContext`. The `EventNotifier` then invokes callbacks during the routing of exchanges. This allows the `NotifyBuilder` to listen for those events and react accordingly. The `EventNotifier` is covered in section 16.3.5.

The `NotifyBuilder` identifies three ways a message can complete:

- *Done*—This means the message is done, regardless of whether it completed or failed.
- *Completed*—This means the message completed with success (no failure).
- *Failed*—This means the message failed (for example, an exception was 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 9.9).

TIP You can create multiple instances of `NotifyBuilder` if you want to be notified of different conditions. The `NotifyBuilder` also supports using binary operations (and, or, not) to stack together multiple conditions.

The source code for the book contains some example of using `NotifyBuilder` in the `chapter9/notify` directory. You can run them using the following Maven goal:

```
mvn test -Dtest=NotifyTest
```

We encourage you to take a look at this source code and also the online documentation.

So far we have covered testing Camel using its own set of test modules, aka the Camel Test Kit. But we do not live in a world of only Camels (although the authors of this book may have too much Camel on their minds), so there is a number of great test libraries for you to leverage. We end this chapter by take a look at how you can use some of these test libraries together with Camel.

9.6 Using third party testing libraries

Over the years a number of great testing frameworks have emerged. We thought they are worth introducing to you and provide some basic examples how you can use them to test your Camel applications. These test frameworks are not a direct replacement for the Camel Test Kit which we have covered extensively in this chapter. These frameworks are suited for integration and system testing, where Camel Test Kit is particular strong for unit testing.

The third party testing libraries being covered are as follows:

- *Arquillian* - Help test applications that run inside a JEE container

- *Pax-Exam* - Help test applications than run inside an OSGi container
- *Citrus Framework* - Automate integration testing for message protocols and data formats
- *Rest Assured* - Java DSL for easy testing of REST services
- *Other Testing Libraries* - A brief list of other kind of testing libraries.

The first three 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. We will use this library in both sections to follow. At the end of the section we quickly cover a number of 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 an JEE server. Readers who have build JEE applications may know that testing your deployments and applications on said servers is not an easy task. The bigger and older the JEE server was, the worse it was. It was not unusual to find yourself developing and testing locally using a lighter JEE server such as JBoss Application Server, and then cross your fingers when testing on a production like system.

The JEE servers has peeked 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 the Arquillian in Action book by Alex Soto Bueno and Jason Porter, published by Manning Publications.

In this book we will show you how to build an integration test with Arquillian that tests a Camel application that runs as a web application on a JEE 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 is defined using XML DSL notation as shown in listing 9.20.

Listing 9.20 Camel RESTful service defined using Rest DSL

```
<bean id="quote" class="camelinaaction.Quote"/> ①

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

    <restConfiguration component="servlet"/> ②

    <rest produces="application/json">
        <get uri="/say">
            <to uri="bean:quote"/> ③
        </get> ④
    </rest> ⑤
```

```
</rest>
</camelContext>

① A bean that returns a quote
② Use the servlet component as the RESTful HTTP server
③ RESTful service that produces json as output
④ GET /say service
⑤ Call the bean to get a quote to return as response
```

The code in listing 9.20 is using Camel's Rest DSL to define a RESTful service. This is a sneak peak of what is coming in the next chapter where REST services is on the menu. The example uses a Java bean that provides with a quote from a wiseman ① . The REST service is using the Servlet component ② to tap into the servlet container of the JEE 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 bean that returns one of the three great wise sayings:

```
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 String.format("{\"quote\": \"%s\"}", QUOTES[idx]);
    }
}
```

The final piece of the application is the web.xml as shown in listing 9.21.

Listing 9.21 web.xml to setup Camel Servlet and bootstrap the Camel XML based application

```
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:camelinaction/quote.xml</param-value> ①
  </context-param>

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>②
  </listener>

  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-
      class>
    <load-on-startup>1</load-on-startup> ③
  </servlet>
```

```

<servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/*</url-pattern>          ④
    </servlet-mapping>

</web-app>

```

- ① Location of the Camel XML DSL file
- ② Use Spring to bootstrap the Camel XML application
- ③ Setup the Camel Servlet
- ④ Use the root path for the Camel Servlet mapping

The web.xml file in listing 9.21 is an old and trusty way of bootstrapping a Web Application. At first we reference the classpath location of the Spring XML file ① that includes the Camel route. Then we use Spring listener ② to bootstrap our application. The Camel Servlet component is configured as a Servlet ③ with an URL mapping ④.

We will 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 its time to pay attention because now Arquillian enters the stage.

SETTING UP ARQUILLIAN

When using Arquillian there are three steps to be taken to create and use Arquillian test.

1. Add Arquillian artifacts to Maven pom.xml
2. Chose 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 its recommended to import the Arquillian Maven BOM in your pom.xml, which makes it easier to setup the right Maven dependencies. This is done by adding the following to the pom.xml file:

```

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.arquillian</groupId>
            <artifactId>arquillian-bom</artifactId>
            <version>1.1.11.Final</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>

```

... and the following dependency as well:

```

<dependency>
    <groupId>org.jboss.arquillian.junit</groupId>
    <artifactId>arquillian-junit-container</artifactId>

```

```
<scope>test</scope>
</dependency>
```

CHOOSING THE APPLICATION SERVER

Because Arquillian runs the tests inside a JEE container you would need to pick which one to use for testing. We used WildFly in section 9.2, so lets use a different container this time, where we chose Jetty by adding the following to the pom.xml file:

```
<dependency>
  <groupId>org.jboss.arquillian.container</groupId>
  <artifactId>arquillian-jetty-embedded-9</artifactId>
  <version>1.0.0.CR3</version>
  <scope>test</scope>
</dependency>
```

Adding the `arquillian-jetty-embedded` dependency does not include Jetty itself, so you need to add the following Jetty dependencies:

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-webapp</artifactId>
  <version>9.2.15.v20160210</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-deploy</artifactId>
  <version>9.2.15.v20160210</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-annotations</artifactId>
  <version>9.2.15.v20160210</version>
  <scope>runtime</scope>
</dependency>
```

Notice how we have set the scope to runtime because Jetty is only used as the runtime JEE container, we do not want to have compile time dependencies on Jetty. We we want our Camel Web application to be portable and run on other JEE containers.

We are now ready to write the integration test.

WRITING THE TEST CLASS WITH THE DEPLOYMENT UNIT AND ACTUAL TEST

We jump straight into code and shows the Arquillian test in listing 9.22.

Listing 9.22 - Arquillian test that creates a deployment unit and test method

```
package camelinaction;

import java.net.URL;
import java.nio.file.Paths;
import org.jboss.arquillian.container.test.api.Deployment;
```

```

import org.jboss.arquillian.container.test.api.RunAsClient;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.arquillian.test.api.ArquillianResource;
import org.jboss.shrinkwrap.api.Archive;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.WebArchive;
import org.junit.Test;
import org.junit.runner.RunWith;
import static com.jayway.restassured.RestAssured.given;
import static org.hamcrest.collection.IsIn.isIn;

@RunWith(Arquillian.class)
public class QuoteArquillianTest {

    @Deployment
    public static Archive<?> createTestArchive() {
        return ShrinkWrap.create(WebArchive.class)
            .addClass(Quotes.class)
            .setWebXML(Paths.get("src/main/webapp/WEB-INF/web.xml").toFile());
    }

    @Test
    @RunAsClient
    public void testQuote(@ArquillianResource URL url) throws Exception {
        given().
            baseUri(url.toString()).
            when().
                get("/say").
            then().
                body("quote", isIn(Quotes.QUOTES));
    }
}

```

- ➊ Run the test with Arquillian
- ➋ Setup the deployment unit in this method
- ➌ Use ShrinkWrap to create a micro deployment unit
- ➍ Run this test as a client
- ➎ Inject the URL of the deployed application
- ➏ Use Rest assured to verify REST call

An Arquillian test is a JUnit test that has been annotated with `@RunWith(Arquillian.class)`

➊ . The deployment unit is defined by a static method that has been annotated with `@Deployment` ➋ . The method creates an Archive using ShrinkWrap where you are in full control of what to include in the deployment unit ➌ . In this example we only need the Quote class and the web.xml file ➍ .

The big picture of ShrinkWrap

ShrinkWrap is used to build the test archive that is deployed into the application server. This allows to build isolated test deployments that only includes what you need. From the application server point of view the ShrinkWrap archive is a real deployment unit.

The Camel application we want to test is using the Camel Servlet component and therefore we want to start the test from the client by calling the servlet, and therefore we need to annotate the test with `@RunAsClient` ④, 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 ⑤.

The implementation of the test method is using the Rest Assured test library that provides a great DSL to define REST based test ⑥. 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 which you can try using the following Maven goal:

```
mvn test
```

There is a lot more to Arquillian than what we can cover in this book. If you are running Camel applications in an application server then we recommend to give Arquillian a try.

And speaking of application servers, lets take a look at a different kind which is the OSGi based Apache Karaf kind.

9.6.2 Using Pax-Exam to test Camel applications

Pax Exam is for OSGi what Arquillian is for JEE servers. If the mountain won't come to Muhammad the Muhammad must go to the mountain. This phrase covers the principle of both Arquillian and Pax Exam. Your tests must go to those big application servers.

Earlier in this chapter in section 9.2.4 we covered testing Camel using OSGi Blueprint with the camel-test-blueprint module. This module has its limitations and there is on so far you can go. When you hit those walls, then you, just as Muhammad, must go the mountain known as Pax Exam.

In this section we will use Pax Exam to test the Camel quote application we built in the last section. When we covered using Arquillian the Camel application was deployed as a Web application, but here with OSGi we deploy the application as an OSGi bundle.

Deploying Web Application to Apache Karaf

You can deploy Web Application in Apache Karaf known as Web bundles. But beware that this is not a common practice and not all web frameworks works on OSGi and you find yourself in an OSGi class loading hell. We do not recommend using this approach and recommend developing Camel applications using either OSGi Blueprint, CDI, or OSGi Declarative Services.

MIGRATING THE CAMEL APPLICATION FROM WEB APPLICATION TO OSGI BUNDLE

We migrate the Camel application from Web Application by:

- Change the pom.xml from war to OSGi bundle
- Add the Felix Bundle Plugin to the pom.xml

- Create a features.xml file to make installing the application easy
- Converting the Spring XML file to OSGi Blueprint XML

We start by changing the pom.xml from web to OSGi by setting packaging to bundle as shown:

```
<packaging>bundle</packaging>
```

Then in the plugins section we add the Maven Bundle Plugin that will generate the OSGi information in the MANIFEST.MF file which turns the JAR into an OSGi bundle.

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>3.0.1</version>
  <extensions>true</extensions>
</plugin>
```

We then add another plugin that attaches the features.xml file to the project as a maven artifact. This ensures the features file gets installed and release 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>
    <feature>camel-blueprint</feature>
    <feature>camel-servlet</feature>
    <bundle>mvn:com.camelinaction/chapter9-pax-exam/2.0.0</bundle>
  </feature>
</features>
```

①
②
③
④

- ① Install the Karaf war feature which includes HTTP servlet support
- ② Install these Camel components
- ③ Install the Camel example itself

The biggest change in the example is the migration from Spring XML to OSGi Blueprint. Its not the actual difference between Spring and Blueprint that is the big changes as these syntax are very similar. Its the fact that we need to setup the Camel Servlet component to use the OSGi HTTP service to tap into the servlet container from the Apache Karaf server. How this is done is shown in listing 9.23.

Listing 9.23 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

<bean class="org.apache.camel.component.servlet.OsgiServletRegisterer"
      init-method="register" ②
      destroy-method="unregister" ②
      <property name="alias" value="/camel"/> ②
      <property name="httpService" ref="httpService"/> ②
      <property name="servlet" ref="quotesServlet"/> ②
</bean>

<bean id="quotesServlet"
      class="org.apache.camel.component.servlet.CamelHttpTransportServlet"/> ③

<bean id="quote" class="camelinaction.Queues"/> ④

<camelContext id="quotesCamel"
               xmlns="http://camel.apache.org/schema/blueprint">

    <restConfiguration component="servlet" ⑤
                       port="8181" contextPath="/camel"/>

    <rest produces="application/json">
        <get uri="/say">
            <to uri="bean:quote"/> ⑥
        </get> ⑦
    </rest>
</camelContext>

</blueprint>
```

- ① Refer to use the OSGi HTTP Service
- ② Setup a Servlet that uses /camel as context-path
- ③ Camel Servlet
- ④ A bean that returns a quote
- ⑤ Use the servlet component as the RESTful HTTP server
- ⑥ RESTful service that produces json as output
- ⑦ GET /say service
- ⑧ Call the bean to get a quote to return as response

The Camel application uses the servlet container provided by Apache Karaf which is done by referring to the OSGi HTTP Service ① . Then we do a bit of XML to setup 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 which you can build and run in Apache Karaf.

At first you need to build the example from Maven using:

```
mvn install
```

Then you can start Apache Karaf and from the Karaf shell type the following commands:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

```
feature:repo-add camel 2.17.1
feature:install camel
feature:repo-add mvn:com.celinaction/chapter9-pax-exam/2.0.0/xml/features
feature:install camel-quote
```

And from a web browser can open the following url:

```
http://localhost:8181/camel/say
```

And you should Camel return a wise phrase such as:

```
{
    quote: "Camel rocks"
}
```

Okay now its time to listen up and cover your hair in your eyes (hint a little tribute to Nirvana performing the MTV unplugged album over 2 decades ago).

SETTING UP PAX EXAM

When using Pax Exam there are two steps to be taken to create and use Pax Exam test.

6. Add Pax Exam artifacts to Maven pom.xml
7. 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 a bunch of artifacts where some are shown below:

```
<dependency>
    <groupId>org.ops4j.pax.exam</groupId>
    <artifactId>pax-exam</artifactId>
    <version>4.8.0</version>
    <scope>test</scope>
</dependency>
<dependency>
<groupId>org.ops4j.pax.exam</groupId>
    <artifactId>pax-exam-junit4</artifactId>
    <version>4.8.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.ops4j.pax.exam</groupId>
    <artifactId>pax-exam-container-karaf</artifactId>
    <version>4.8.0</version>
    <scope>test</scope>
</dependency>
```

You can find the full list of the Pax Exam Maven artifacts in the source code for this example which is located in the chapter9/pax-exam directory.

WRITING THE TEST CLASS WITH THE DEPLOYMENT UNIT AND ACTUAL TEST

Lets jump straight into code and follow up explaining the important details.

Listing 9.24 - Pax Exam Test Class with Deployment and actual test

```

@RunWith(PaxExam.class)          ①
public class PaxExamTest {

    @Inject
    @Filter("(camel.context.name=quotesCamel)")      ②
    protected CamelContext camelContext;

    @Configuration                                ③
    public Option[] config() {
        return new Option[]{
            karafDistributionConfiguration()          ④
                .frameworkUrl(maven().groupId("org.apache.karaf")
                    .artifactId("apache-karaf").version("4.0.5").type("tar.gz")) ④
                .karafVersion("4.0.5")                  ④
                .name("Apache Karaf")                 ④
                .useDeployFolder(false)               ④
                .unpackDirectory(new File("target/karaf")),           ④

            keepRuntimeFolder(),                   ⑤
            configureConsole().ignoreRemoteShell(),       ⑥
            logLevel(LogLevelOption.LogLevel.WARN),         ⑥

            junitBundles(),
            features(getCamelKarafFeatureUrl(), "camel", "camel-test"), ⑦
            features(getCamelKarafFeatureUrl(), "camel-http"),          ⑧
            features(maven().groupId("com.camelinaction")
                .artifactId("chapter9-paxexam").version("2.0.0")        ⑨
                .classifier("features").type("xml"), "camel-quote")      ⑨
        };
    };                                         ⑩

    public static UrlReference getCamelKarafFeatureUrl() {
        return mavenBundle().
            groupId("org.apache.camel.karaf").artifactId("apache-camel")
            .version("2.17.1").classifier("features").type("xml");
    }

    @Test
    public void testPaxExam() throws Exception {
        long total = camelContext.getManagedCamelContext().getExchangesTotal();
        assertEquals("Should be 0 exchanges completed", 0, total);

        String url = "http://localhost:8181/camel/say";
        ProducerTemplate template = camelContext.createProducerTemplate();
        String json = template.requestBody(url, null, String.class);   ⑪
        System.out.println("Wiseman says: " + json);

        total = camelContext.getManagedCamelContext().getExchangesTotal();
        assertEquals("Should be 1 exchanges completed", 1, total);
    }
}

```

- 1 Run this test with Pax-Exam
- 2 Inject the CamelContext from the application
- 3 Setup the deployment unit
- 4 Configure the Apache Karaf server to use
- 5 Keep the runtime folder so we can look there in case of errors
- 6 Set the logging level to not be verbose
- 7 Install Apache Camel
- 8 Install the camel-test component we use in this test
- 9 Install this example using the Maven coordinates
- 10 The actual test method that runs inside Karaf container
- 11 Call the Camel Servlet and print the response

You create a Pax Exam test by annotating the test class with `@RunWith(PaxExam.class)` ① . You can do dependency injection using `@Inject` which is used to inject the `CamelContext` of the Camel application being tested ② . Then we configure what to deploy and run on the Apache Karaf server in the `@Configuration` method ③ . At first we configure the Apache Karaf server to use ④ which will be unpacked in the `target/karaf` directory, and started from there. Its 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 very verbose in the logging, so we turn 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 we install Apache Camel ⑦ and the `camel-http` component ⑧ which we used for testing our Camel application that is installed next ⑨ .

The remainder of the code is the actual test method ⑩ which uses the injected `CamelContext` ⑪ to create a `ProducerTemplate` which we use to call the Camel servlet ⑫ . Notice how we assert that Camel has processed no message before the test, and then one message afterwards. In the source code example we also have addition test to verify the returned message is one of the expected quotes. But we left that code out from the listing.

RUNNING THE TEST

Now its a good time to running the Pax Exam test yourself from the `chapter9/pax-exam` directory. At first you need to build the example without testing:

```
mvn clean install -Dtest=false
```

And then you can run the test with:

```
mvn test
```

You may ask yourself why do I need to build the example without testing first? The reason is a catch-22 situation. The Pax Exam test is installing the Camel application using a features file using the maven coordinates ⑨ , and the features file is only installed into your local Maven repository from the `install` goal, which by default runs after test.

First time users with Pax Exam will have a mouthful to learn and master as setting up the `@Configuration` method can be tricky. Pax Exam has many options and there is more to it

than what we can cover in this book. But we have shown how you can get on a good start with testing Camel applications using Pax Exam.

Let's leave OSGi and try a different kind of fruit, a citrus fruit.

9.6.3 Using Citrus Framework to test Camel applications

The Citrus Framework is a test framework for integration testing. Enterprise software typically take part where data are exchanged between different systems and/or business partners. The data being exchanged across those interfaces is essential to perform as expected and according to agreed upon data formats and messaging protocols.

Testing those scenarios to ensure correct data communication and payload and as well that the systems handles both successful and failure use cases, is the scope of integration testing. Citrus is a test framework that is intended to help you become successful with integration testing. You may picture this as what Camel does for integration Citrus does for integration testing. Figure 9.7 illustrates the principle how Citrus works.

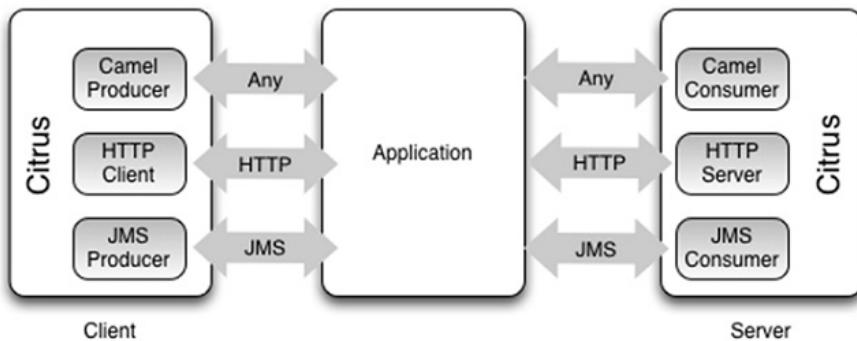


Figure 9.7 In a typical test scenario the application being tested is deployed on an application server and interacts with Citrus over various message transports. During the test run Citrus is able to act on both sides as client and/or server simulating request/response messages.

As you can see from figure 9.7 then the application you are testing is tested by Citrus that acts as a client. Citrus interact with the application by sending in messages using various protocols that is supported by Citrus. In case there is no direct support from Citrus, then there is Camel where you can use any of the Camel components for the communication with the application. The application may need to communicate with other systems, and Citrus can act as a server to simulate those systems where you can define what messages the server should response. With each test step you can validate the exchanged messages with expected control data. The test is fully automated and repeatable, so you can add integration tests to your continuous build.

INTEGRATING THE NEW WITH THE OLD

At Rider Auto Parts the order system is a legacy system that only provides access using a Java JMS interface. You have been running a pilot project to build a quick prototype that facades the legacy system with a micro service that provides a RESTful service to query order status. This micro service is a key in a puzzle at the company to build a next generation mobile and web application for your customers. Currently you are still dipping your toes with changing to this new paradigm with micro services, 12-factor applications, and what gives. But what seems to prevail is the legacy systems, with the ERP system running on IBM AS/400 mainframe.

Figure 9.8 shows what you are building.

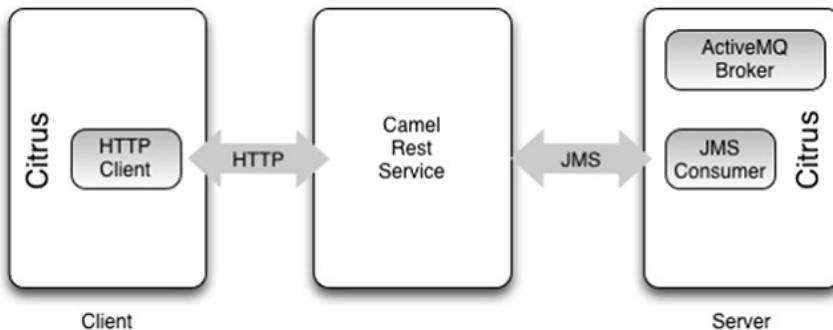


Figure 9.8 Using Citrus as both client and service to perform integration tests of the Camel REST service application. As client Citrus uses HTTP to call the Camel application. As server Citrus simulates the backend system by embedding an ActiveMQ broker.

You built the Camel prototype quickly using the following lines of XML:

Listing 9.25 - Camel microservice exposes a REST service that routes to the legacy system using JMS

```

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

<restConfiguration component="jetty" port="8080" contextPath="/order"/> ①

<rest produces="text/xml">
    <get uri="/status">
        <to uri="direct:status"/>
    </get>
</rest>

<route>
    <from uri="direct:status"/>
    <transform>
        <simple><![CDATA[<order><id>${header.id}</id></order>]]></simple> ③
    </transform>
</route>

```

```

<to uri="jms:queue:order.status"/>
</route>

</camelContext>

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

- ➊ Configure a REST service to use Jetty on port 8080
- ➋ REST GET /status service
- ➌ Transform message to XML with the order id provided
- ➍ Call the legacy system using JMS bridge
- ➋ Setup Camel JMS component to use the JMS ConnectionFactory

You have peaked ahead in this book and read parts of chapter 10 that covers the Camel Rest DSL that allows to define RESTful services using a DSL ➊ ➋ . The prototype is using the Camel Jetty component on port 8080 to host the RESTful service ➌ . The service has one GET method on uri `/order/status` ➋ that routes to a Camel route using the direct component. In this route the message is transformed to XML format with the provided order id, that comes from HTTP query parameter that Camel has mapped to a header ➌ . The order status is then obtained using JMS messaging ➍ using request/reply. The legacy system is using IBM WebSphereMQ as the message broker, and you have setup the Camel JMS component to refer to a JMS ConnectionFactory ➋ to be used. This allows you to plugin different providers such as using ActiveMQ for testing.

Lets use Citrus to test the prototype you have built.

SETTING UP CITRUS IN MAVEN

At first you need to add some dependencies and Maven plugins to your Maven pom.xml file:

- Add the Citrus dependencies
- Add Citrus Maven plugin
- Add the Maven Fail-Safe Maven plugin

You have added the following Maven dependencies:

```

<dependency>
    <groupId>com.consol.citrus</groupId>
    <artifactId>citrus-core</artifactId>
    <version>${citrus.version}</version>
</dependency>
<dependency>
    <groupId>com.consol.citrus</groupId>
    <artifactId>citrus-java-dsl</artifactId>
    <version>${citrus.version}</version>
</dependency>
<dependency>
    <groupId>com.consol.citrus</groupId>
    <artifactId>citrus-camel</artifactId>
    <version>${citrus.version}</version>
</dependency>
```

```
<dependency>
    <groupId>com.consol.citrus</groupId>
    <artifactId>citrus-http</artifactId>
    <version>${citrus.version}</version>
</dependency>
<dependency>
    <groupId>com.consol.citrus</groupId>
    <artifactId>citrus-jms</artifactId>
    <version>${citrus.version}</version>
</dependency>
```

The Citrus Maven Plugin is added and you refer to the root package where your source code are in the targetPackage configuration as shown:

```
<plugin>
    <groupId>com.consol.citrus.mvn</groupId>
    <artifactId>citrus-maven-plugin</artifactId>
    <version>${citrus.version}</version>
    <configuration>
        <targetPackage>camelinaction</targetPackage>
    </configuration>
</plugin>
```

And finally you add the Failsafe Maven Plugin. Yes the plugin is named like that, but its usage is to run integration tests using the integration-test goal:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>2.19</version>
    <executions>
        <execution>
            <id>integration-tests</id>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Then we need to setup Citrus and write an integration test.

SETTING UP CITRUS

Citrus Framework was created when Spring XML configuration was a very popular choice for configuring your applications. Today it's still a good choice for setting up Citrus which you do in the citrus-context.xml file as shown in listing 9.26.

Listing 9.26 Setting up Citrus to embed an ActiveMQ broker and provide JMS and HTTP endpoints which are for testing.

```

<context:property-placeholder location="classpath:citrus.properties"/> ①

<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="${jms.broker.url}"/>
   ②

<amq:broker useJmx="false" persistent="false">
  <amq:transportConnectors>
    <amq:transportConnector uri="${jms.broker.url}"/>
  </amq:transportConnectors>
</amq:broker>

<citrus-jms:sync-endpoint id="statusEndpoint"
  connection-factory="jmsConnectionFactory"
  destination-name="order.status" timeout="10000"/> ④

<citrus-http:client id="statusHttpClient"
  request-method="GET" request-url="http://localhost:8080/order"
  content-type="text/xml" timeout="60000"/> ⑤

<import resource="classpath:camelinaction/status.xml"/> ⑥

```

- ① Use property placeholders from citrus.properties
- ② Setup JMS ConnectionFactory to use ActiveMQ
- ③ Embed ActiveMQ as JMS broker used during testing
- ④ Citrus JMS endpoint used for simulating legacy system
- ⑤ Citrus HTTP client used for testing as a client
- ⑥ Load the Camel XML file

The citrus-context.xml file is a Spring XML file with `<bean>`s and some Citrus specific configuration. In the top of the XML file you have 14 lines to import and setup namespaces, which has been omitted from listing 9.26. You can use property placeholders using Spring ① which is used to setup the URL ② to the embedded ActiveMQ Broker ③. In this Citrus integration test we use Citrus to simulate the legacy backend by setting up a Citrus JMS endpoint ④ using the queue name `order.status`. To call the Camel application we use Citrus HTTP client which we also setup ⑤. And finally we tell Citrus to import the Camel XML file ⑥.

Now lets write the Citrus Integration test source code.

WRITING THE CITRUS TEST CODE

You write Citrus Integration tests in either XML or Java code. We are already using Spring XML for the Camel application and setting up Citrus, so lets use earn our salary and write the integration test using a programming language such as Java.

Listing 9.27 shows the source code with a full Citrus integration test, which tests your Camel prototype.

Listing 9.27 Citrus Integration test the Camel prototype

```
public class CitrusIT extends JUnit4CitrusTestDesigner { ①

    @Test @CitrusTest
    public void orderStatus() throws Exception {

        description("Checking order should be ON HOLD");
        variable("orderId", "citrus:randomNumber(5)"); ③

        http().client("statusHttpClient")
            .get("/status?id=${orderId}") ④
            .contentType("text/xml").accept("text/xml")
            .fork(true); ⑤

        echo("Sent HTTP Request with orderId: ${orderId}");

        receive("statusEndpoint") ⑥
            .payload("<order><id>${orderId}</id><status>ON HOLD</status></order>")
            .extractFromHeader(JmsMessageHeaders.CORRELATION_ID, "cid"); ⑦

        send("statusEndpoint") ⑧
            .payload("<order><id>${orderId}</id><status>ON HOLD</status></order>")
            .header(JmsMessageHeaders.CORRELATION_ID, "${cid}");

        http().client("statusHttpClient") ⑨
            .response(HttpStatus.OK)
            .payload("<order><id>${orderId}</id><status>ON HOLD</status></order>")
            .contentType("text/xml");
    }
}
```

- ① Extend test class to use the Citrus Java DSL to design the test
- ② Specify test method is using Citrus
- ③ Setup random order number with 5 digits
- ④ Call the Camel application using HTTP GET
- ⑤ Fork the call as we need to setup more before the reply comes back
- ⑥ Expect to receive a XML message on the JMS endpoint
- ⑦ Remember the JMS CorrelationID
- ⑧ Send back JMS reply message using the expected JMSCorrelationID
- ⑨ The HTTP client is expected to receive a XML response with order status ON HOLD

Because we are using Citrus Java DSL to design the integration test we must extend the test class with Citrus's `JUnit4CitrusTestDesigner` class. The test method must then be annotated with `@CitrusTest`. It can be beneficial to use random data during integration testing that helps avoiding using hardcoded test values that may cause shielding from test failures. So we use the random function from Citrus to generate a 5 digit order number which we store in a variable ③.

We use Citrus to act as the client calling the Camel RESTful service using HTTP with the random order number ④. Its important to use `fork(true)` ⑤ so Citrus do not block and execute the HTTP client call and wait for any repose before continuing. We want to continue designing the integration test as we need to simulate the legacy system by tapping into JMS and listen on the JMS queue for the expected message to arrive ⑥. To ensure we respond

with the correct `JMSCorrelationID` we need to store that as a variable in Citrus ⑦ . We then tell Citrus what message to send back as the JMS reply message ⑧ . Notice how we can use the variables `${orderId}` and `${cid}` to refer to the order id and correlation id. Finally we tell Citrus what the HTTP client is expected to receive as response message ⑨ .

RUNNING THE INTEGRATION TEST

The source code for the book contains this example in the chapter9/prototype directory which you can try using the following Maven goal:

```
mvn integration-test
```

Yes you have to type `integration-test` as the test goal is only for running unit tests.

That is all we could squish out of the citrus for this book. The Citrus Framework has more to offer and we encourage you to take a look at this framework if you are building sophisticated integration tests.

The last section is a brief section that quickly highlight six different testing libraries that are phenomenal in each of their own way.

9.6.4 Other Testing Libraries

Testing has many facets and in this book we have chose to focus on unit and integration testing with Camel which are disciplines that has the strongest coupling to Camel. Other disciplines such as load and performance tests are more general applicable and you can find other resources that covers those concepts very well. We do have list of noteworthy testing libraries we wanted to bring to your attention.

GATLING

Gatling (<http://gatling.io>) is a load testing library that provides a DSL which allows you to write load testing at a high abstraction level. The library also has a recorder tool that allows you to use a browser to capture what you do, and then generate this as DSL code you can use to run these tests over and over again. Gatling also provides beautiful and informative reports as output of running the tests.

BYTEMAN

Byteman (<http://byteman.jboss.org>) is a bytecode manipulation tool that can inject custom code into specific points. For example you can use this to inject code that triggers errors to happen at certain places that otherwise would be hard to cause to happen without using byteman.

JAVA MISSION CONTROL

Java Mission Control is part of Java which you can run from a shell using jmc. JMC is able to collect low level detail about running Java applications with a very little overhead (1% or less). With the tool you can visually analyze the collected data, and help understand how your application is behaving on the JVM.

JHICCUP

JHiccup (<https://github.com/giltene/jHiccup>) is a tool that logs and records JVM platform "hiccups" such as JVM stalls that happens during garbage collection runs, or hardware platform noise that otherwise will cause your application to not be constantly running. The tool provides reports where you can identify those hiccups as spikes in the graphs.

FLAMEGRAPH

Flamegraph (<http://www.brendangregg.com/flamegraphs.html>) are a visualization of an entire software stack, where the most frequent code paths is easily identified. The tool outputs a graphical visualization that resemble flaming fire and hence the name.

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 setup a new Java project which can be created using a Maven Archetype and then 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 then easily specify to run tests in intervals of tens or thousands and also allow the JVM to warmup first etc.

PERFCAKE

PerfCake (<https://www.perfcake.org>) is a lightweight performance testing framework. The framework also includes a load generator that allows you to script test cases that tests your systems under heavy load. After running the tests you can analyze the results as raw data or from visual graphs.

That's it folks. We have reached the end of what we have to say. All that is left is the chapter summary.

9.7 Summary and best practices

Testing is a challenge for every project. It's generally considered bad practice to only do testing 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 only do once, either. Testing is involved in most phases in a project. You should do unit testing while you develop the application. And you should also

implement integration testing to ensure that the different components and systems work together. You also have the challenge of ensuring you have the right environments for testing.

Camel can't eliminate these challenges, but it does provide a great Test Kit that makes writing tests with Camel applications easier and less time consuming. We looked at this Test Kit. You can run Camel in any kind of Java environment you like. We covered how to write Camel unit tests for most popular ways of using Camel, whether its running plain Camel standalone, with Spring in any form or shape, with CDI, OSGi, or on popular JEE servers such as WildFly.

We also reviewed how you can 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 will learn all about error handling, and in this chapter you saw how you can 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, where mocks are a real advantage. In integration testing, mocks aren't available to use, and 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).

Towards the end of this chapter we covered testing Camel with a number of integration testing libraries that makes it possible to write tests that deploys and runs in the application server of choice. We saw how Arquillian makes it easier to run tests inside a JEE server such as WildFly. For OSGi users we covered how to use Pax Exam for running tests inside Apache Karaf containers. For the juicy users we learned how to get started with writing integration tests using Citrus to test your Camel applications.

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 test(s).
- *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 application server.* If you run your Camel applications on an application server, we recommend to use Arquillian or Pax Exam to run tests that actually deploy and run on the application server.

- *Testing REST services.* The third party testing library Rest Assured is a great tiny library for writing human understandable unit tests that calls REST services. Just as Camel it provides a DSL that almost speak in human terms what is happening. Make sure to pickup this library if you jump on the hype of REST and JSON.
- *Do not neglect integration tests.* Build integration tests that tests that the individual components work together. You can get far by using the Camel Test Kit for integration testing. But we encourage you to look at using third party frameworks such as Arquillian and Citrus Framework.
- *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. Setup a continues integration platform that performs testing on a regular basis, and allows instant feedback loop to your development team.

And speaking of REST this was a great segue to the next chapter is also a long chapter that covers all about modern RESTful web services and the old school SOAP based web services.

10

REST and Web Services

This chapter covers

- Restful services fundamentals
- Building REST services with JAX-RS and Apache CXF
- Building REST services with camel-restlet and camel-cxf component
- Using Camel's Rest DSL
- How the Rest DSL works and supported components
- Using Rest DSL with various Camel components
- Binding between POJOs and XML/JSON
- Documenting your REST services using Swagger
- Browsing Swagger API using Swagger UI
- Web services using Apache CXF
- Developing *contract-first* web services
- Developing *code-first* web services

For over a decade Web services has played an important role in integration loosely and disparate systems together. As a natural evolution of the human kind and the IT industry we learn and get smarted. In recent time REST services has taken over the role as first choice for integrating services. This chapter covers a lot of ground how to develop and use REST and Web Services with Camel. We have focused on the present and future and spend 80% on REST and 20% on Web Services.

Section 10.1 covers the fundamentals and principles about restful services, so you are ready to tackle the 80% content of this chapter. We have provided a lot of examples with source code that focus on a simple use-case that could have happened in the real world, if

Rider Auto Parts was an actual company. This section continues as a journey to tell you how you can implement the REST services using different REST frameworks and Camel components.

In section 10.2 this section introduces you to the Rest DSL which allows you to define REST services *the Camel way*, where 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 using Swagger APIs, that allows consumers to easily obtain the contracts of your services.

The last 15% of this chapter is dedicated to the old world of SOAP based Web Services where we cover how you can develop contract and first based web services with Apache CXF and Camel.

Okay are you ready? Let's start with the new and existing stuff around Restful services.

10.1 Restful services

Restful services or also known as REST has become a very popular architectural style used in modern enterprise projects. REST was defined by Roy Fielding in 2000 when he published his paper (according to wikipedia) and today REST is a foundation that drives the modern APIs on the web. You can also think this as a modern web services, where the APIs are RESTful and HTTP based so they are easily consumable on the web. In the 2nd half of this chapter we will take a look at the predecessor to REST services, which is SOAP based web services.

In this section we will use an example from the Rider Auto Parts company to explain the principle of an RESTful API, and then move on to cover how you can implement this service using the following REST frameworks and Camel components:

- *Apache CXF* - Shows how to use only CXF to build REST services
- *Apache CXF with Camel* - Previous example with Camel included
- *Camel Restlet component* - Using one of the simplest Camel Rest components
- *Camel CXF REST component* - Using Camel with CXF REST component

Before we start the show let's cover the basic principles of what a RESTful service is.

10.1.1 The principle of an RESTful API

At Rider Auto Parts you have 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 are a devoted developer, and with the limited resources the company offered, you had to take matters in your own hand, and over a weekend you sketched the RESTful API design, which is shown 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/123 |
|--------|---|---|
| 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 the HTTP protocol at the base path <http://rider.com/orders>. REST services reuse the same HTTP verbs that are already well understood and established from the HTTP protocol. The GET verb is like a SQL query function to access and return data. The PUT/POST verbs is similar to SQL INSERT or UPDATE functions that creates or updates data. And finally we have the DELETE verb which is for deleting data.

To allow legacy systems to access the service you decided 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

JAX-RS (Java API for RESTful Web Services) is a Java standard API that provides support for creating web services. There are a number of REST frameworks that implements the JAX-RS specification such as Apache CXF. JAX-RS is dominantly developed with the help of a set of Java annotations that you use in Java classes that represent the web services.

Before we dive into JAR-RS then lets simplify the use-case from Rider Auto Parts, by removing two of the services from table 10.1, which are not necessary, which gives us the four services listed in table 10.2.

Table 10.2 The first set of RESTful API for the Rider Auto Parts order web service

| Verb | http://rider.com/orders | http://rider.com/orders/123 |
|--------|---|---|
| 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 |

We can then map the services from table 10.2 to a Java interface as show:

```
public interface OrderService {
    Order getOrder(int orderId);
```

```

    void updateOrder(Order order);

    String createOrder(Order order);

    void cancelOrder(int orderId);
}

```

And create a Java model class to represent the order details as show:

```

@XmlRootElement(name = "order")
@XmlAccessorType(XmlAccessType.FIELD)
public class Order {
    @XmlAttribute
    private int id;
    @XmlAttribute
    private String partName;
    @XmlAttribute
    private int amount;
    @XmlAttribute
    private String customerName;

    // getter/setter omitted
}

```

The model class is annotated with JAXB making message translation between Java and XML/JSON much easier, because frameworks such as JAXB and Jackson knows 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 listing 10.1.

JAX-RS implementation of the Rider Auto Parts REST service

```

import javax.ws.rs.DELETE;                                ①
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/")
@Produces("application/xml")
public class RestOrderService {                          ②

    private OrderService orderService;

    public void setOrderService(OrderService orderService) { ③
        this.orderService = orderService;
    }

    @GET
    @Path("/{id}")
    public Response getOrder(@PathParam("id") int orderId) { ④
        Order order = orderService.getOrder(orderId);
    }
}

```

```

    if (order != null) {
        return Response.ok(order).build();
    } else {
        return Response.status(Response.Status.NOT_FOUND).build();
    }
}

@PUT
public Response updateOrder(Order order) {           ⑤
    orderService.updateOrder(order);
    return Response.ok().build();
}

@POST
public Response createOrder(Order order) {           ⑥
    String id = orderService.createOrder(order);
    return Response.ok(id).build();
}

@DELETE
@Path("/{id}")
public Response cancelOrder(@PathParam("id") int orderId) { ⑧
    orderService.cancelOrder(orderId);
    return Response.ok().build();
}
}

```

- ① Java import the JAX-RS annotations
- ② Define the entry path of the RESTful web service
- ③ To inject the OrderService implementation
- ④ GET order by id
- ⑤ The Response builder to return the order model in the response
- ⑥ PUT to update an existing order
- ⑦ POST to create a new order
- ⑧ DELETE to cancel an existing order

At first the class imports a number of JAX-RS annotations ① . A class is defined as a JAX-RS resource class by annotating the class with `@Path` ② . 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 being with `/orders/`. The class is also annotated with `@Produces("application/xml")` ② which declares that the service outputs response as XML. We will later see how you can use JSON or support both.

Because we want to separate the JAX-RS implementation from our business logic, we use dependency injection to inject the `OrderService` instance ③ which holds the business logic.

The service that gets order by id is annotated with `@Path("/{id}")` ④ and as well in the method signature using `@PathParam("id")`. This means the parameter is mapped to the context-path with the given key. So for example if a client calls the rest service with `/orders/123`, then 123 is resolved as the id path parameter, which will be mapped to the corresponding method parameter.

If the service returns an order then notice how the response builder includes the order instance when the ok response is being created ⑤ . And likewise if no order could be found, then response builder is used to create a HTTP not found error response.

The following two services to update ⑥ and create ⑦ an order are simple two lines of code. Though notice that these two methods uses the Order type as parameter. And because we have annotated the Order type with JAXB annotations, then Apache CXF will automatically map the incoming XML payload to the Order POJO class.

The last method to cancel an order by id ⑧ , uses simple path parameter mapping as the get order by id service.

The source code for the book contains this example in the chapter10/cxf-rest directory, which you can try using the following Maven goal:

```
mvn compile exec:java
```

The example comes with two dummy orders included, which makes it easier to try the example. For example to get the first order you can from a web browser open the url:

```
http://localhost:9000/orders/1
```

The second order is no surprisingly with the order id 2, so you can get it using:

```
http://localhost:9000/orders/2
```

Testing REST services from web browser

Testing REST services from a web browser is only easy when testing GET services as those can be accessed by typing the url in the address bar. However to try POST, PUT, DELETE and other REST verbs is much harder. Therefore you can install 3rd party plugins that provides a REST client. On Google Chrome there is for example the popular postman plugin.

USING JSON INSTEAD OF XML

Apache CXF allows to plugin different binding providers, which allows to plugin 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>cxfrt-rs-extension-providers</artifactId>
    <version>3.1.5</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.jaxrs</groupId>
    <artifactId>jackson-jaxrs-json-provider</artifactId>
    <version>2.7.1</version>
</dependency>
```

Sadly CXF does not support auto discovering the provides from the classpath, so you need to enlist the Jackson provider on the CXF RS server as shown below:

```
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
sf.setProvider(JacksonJsonProvider.class);
```

The source code for the book contains this example in the chapter10/cxf-rest-json directory which you can try using the following Maven goal:

```
mvn compile exec:java
```

What we have covered in this section was using Apache CXF without Camel, and because this is a book about Apache Camel, its time we get back on the saddle and ride the Camel. So lets see how we can add Camel to the Rider Auto Parts order service application in the sections to follow. But first lets summaries the pros and cons of using a pure Apache CXF model for REST services.

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 is not 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 non trivial |
| There is no CXF in Action book | |

Okay lets 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 using JAX-RS with CXF. However you know that down the road that changes are in the pipeline that requires integration with a number of other systems, and for that you want Camel to play its part. So how can we developed REST services using the JAX-RS standard and still use Camel? One way would be to let CXF integrate with Camel.

To use Camel from any existing Java application can be done in many ways, possible one of the easiest is to use Camel's `ProducerTemplate` from the Java application to send messages to Camel. Listing 10.2 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) { ❶
        this.producer = producerTemplate;
    }

    @GET
    @Path("/{id}")
    public Response getOrder(@PathParam("id") int orderId) {
        Order order = producer.requestBody("direct:getOrder",
                                            orderId, Order.class); ❷
        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); ❸
        return Response.ok().build();
    }

    @POST
    public Response createOrder(Order order) {
        String id = producer.requestBody("direct:createOrder",
                                         order, String.class); ❹
        return Response.ok(id).build();
    }

    @DELETE
    @Path("/{id}")
    public Response cancelOrder(@PathParam("id") int orderId) {
        producer.sendBody("direct:cancelOrder", orderId); ❺
        return Response.ok().build();
    }
}

```

- ❶ To inject Camel's ProducerTemplate
- ❷ Calling Camel route to get the order by id
- ❸ Calling Camel route to update order
- ❹ Calling Camel route to create order
- ❺ Calling Camel route to cancel order

The code in listing 10.2 looks similar to the code from listing 10.1 with the difference that Camel's ProducerTemplate ❶ will be used to route messages from the REST web service to a Camel route ❷ ❸ ❹ ❺ . The Camel route is responsible for routing the message to the Rider Auto Parts backend system giving access to the order details.

Because we essentially have both CXF and Camel independently operating in the same application, we need to setup them individually and configure them in the right order. To keep

things simple you decided to run the application as a standalone Java application with a single main class to bootstrap the application as shown in listing 10.3.

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();
        route.setOrderService(new BackendOrderService()); 1

        CamelContext camel = new DefaultCamelContext(); 2
        camel.addRoutes(route);

        ProducerTemplate producer = camel.createProducerTemplate(); 3

        RestOrderService rest = new RestOrderService();
        rest.setProducerTemplate(producer); 4

        JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean(); 5
        sf.setResourceClasses(RestOrderService.class);
        sf.setResourceProvider(RestOrderService.class,
            new SingletonResourceProvider(rest));
        sf.setProvider(JacksonJsonProvider.class);
        sf.setAddress("http://localhost:9000/"); 6 7

        Server server = sf.create();

        camel.start(); 8
        server.start(); 9

        // code that keeps the JVM running omitted

        camel.stop(); 10
        server.stop();
    }
}
```

- 1 Camel route that access the backend order system
- 2 Create CamelContext and add the route
- 3 Create ProducerTemplate
- 4 Inject ProducerTemplate to REST web service
- 5 Setup CXF REST web service
- 6 Use Jackson for JSON support
- 7 The URL for published REST web service
- 8 Start Camel and CXF
- 9 Keep the JVM running until its stopped
- 10 Stop Camel and CXF

The first part of the application is to setup Camel by creating the route 1 used to call the backend system. Then Camel itself is created by creating a `CamelContext` 2 where the route is added. To integrate Camel with CXF we are creating a `ProducerTemplate` 3 that gets injected into the REST resource class 4. After setting up Camel its CXF turn. To use a JAX-RS server in CXF you create and configure that using a `JAXRSServerFactoryBean` 5. Support

for JSON is done 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 then started by starting Camel and CXF in that order [8](#). To keep the application going there is some code that keeps the JVM running until it's signaled to stop [9](#). To let the application stop gracefully CXF and Camel is being stopped [10](#) before the JVM terminates.

TIP You can also setup CXF REST web services using XML configuration which we will cover later in this chapter.

This example is shipped as part of the source code for the book which you can find in the chapter10/cxf-rest-camel directory, which can be run using the following Maven goal:

```
mvn compile exec:java
```

By combining CXF with Camel gives you best of two worlds. You can define and code the JAX-RS REST services using the JAX-RS standard in Java code, and still integrate Camel using a simple dependency injection technique to inject a `ProducerTemplate` or Camel proxy. However it comes with the cost of you having to write this in Java code, and manually have to control the lifecycle of both CXF and Camel. This can be summarized as 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 non trivial |
| Camel is integrated using dependency injection into the JAX-RS resource class | There is no CXF in Action book |
| The REST service allows developers full control of what happens as they can change the source code | Need to setup both CXF and Camel lifecycle separately |

We will later see how you can use CXF REST services from a pure Camel approach where Camel handles the lifecycle of CXF using the camel-cxf component. We will leave CXF for a moment and look at alternative REST libraries you can use. For example how the service can be implemented without JAX-RS and only use Camel with the camel-reslet component.

10.1.4 Using camel-reslet with REST services

The camel-reslet component is one of the easiest component to use for hosting REST services as Camel routes. The component has also been part of Apache Camel for a very long time, in fact the component was included at end of 2008. The component is easy to use, so lets see that in action.

To implement the services from table 10.2 as REST service with camel-restlet, all needed to be done is to define each service as a Camel route and call the order service bean which performs the action, as shown in listing 10.4.

Listing 10.4 - Camel route using camel-restlet to define 4 REST services

```
public class OrderRoute extends RouteBuilder {

    public void configure() throws Exception {
        from("restlet:http://0.0.0.0:8080/orders?restletMethods=POST")           ①
            .bean("orderService", "createOrder");

        from("restlet:http://0.0.0.0:8080/orders/{id}?restletMethods=GET")          ②
            .bean("orderService", "getOrder(${header.id})");

        from("restlet:http://0.0.0.0:8080/orders?restletMethods=PUT")                ③
            .bean("orderService", "updateOrder");

        from("restlet:http://0.0.0.0:8080/orders/{id}?restletMethods=DELETE")         ④
            .bean("orderService", "cancelOrder(${header.id})");
    }
}
```

- ① REST service to create order
- ② REST service to get order by id
- ③ REST service to update order
- ④ REST service to cancel order by id

The REST web services in listing 10.4 is defined with a route per REST operation, so there is four routes in total. Notice how 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 to map dynamic values from the context-path using the `{ }` style, which we use to map the `id` from the context-path to a header on the Camel message, as illustrated in figure 10.1.

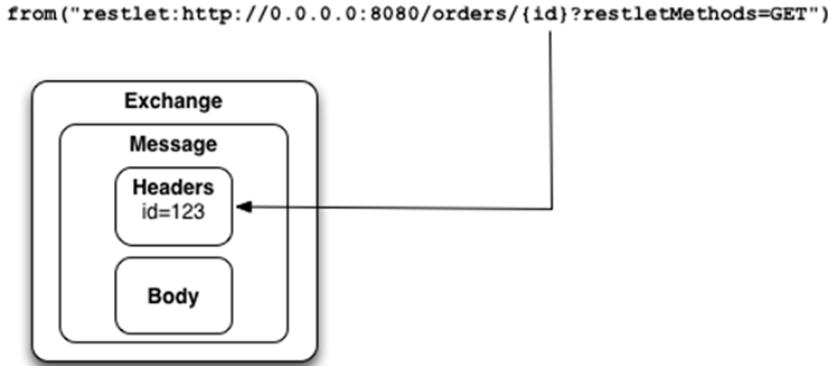


Figure 10.1 Mapping of dynamic values from context-path to Camel Message headers with the camel restlet component

In figure 10.1, the REST web service to get order by id from listing 10.4 is shown in 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

The routes in listing 10.4 is using the normal Java DSL syntax where routes begins with from and routes to bean using bean or to etc. We will later in this chapter learn about the REST DSL that allows us to define REST web services using REST verbs instead of the EIP verbs.

The source code for the book contains this example in the chapter10/restlet directory, which you can try using the following Maven goal:

```
mvn test -Dtest=OrderServiceTest
mvn test -Dtest=SpringOrderServiceTest
```

The restlet component performs no automatic binding between XML or JSON to Java objects, and therefore you would 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 dependency.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

What happens is that by adding camel-jaxb to the classpath, then a *fallback type converter* is added to Camel's type converter registry. A fallback type convert is capable of converting between multiple types decided at runtime, whereas regular type converters can only convert from 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 then 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 what camel-jackson does. However when using camel-jackson a few extra steps are required:

- Enable camel-jackson as a fallback type converter
- Optional annotate the POJOs with JAXB annotations
- Optional annotate the POJOs with Jackson annotations

Both camel-jaxb and camel-jackson are able to use the same set JAXB annotations, which allows you to customize how the mapping to your POJOs should happen. However Jackson is able to perform mapping to any POJOs if the POJO has plain getter/setters, and the JSON payload matches the names of those getter/setter methods. However you can use JAXB to further configure the mapping, and in addition Jackson provides a number of annotations on top of that. You can find more details in 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 its properties on the CamelContext. From Java code you would 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 setup your Camel routes. When using XML you need to configure this inside <camelContext> as shown:

```
<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 which you can try 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 (eg as a producer). As an example see the source code of the chapter10/restlet-json example where the unit tests uses a ProducerTemplate with a camel-restlet uri to call the REST web service.

The camel-restlet component was one of the first Camel component to integrate with REST. The component is easy to get started with and requires little configuration to get going. The pros/cons of using camel-restlet has been summarized in table 10.5.

Table 10.5 Pros and cons of using a camel-restlet

| Pros | Cons |
|---------------------------------------|---|
| Easy to get started | Do not use the JAX-RS standard |
| Restlet is a established REST library | camel-restlet has not been integrated with gson or jackson to make using json easy. |

Lets take a look at one more REST library before we end this section. We 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 include support for both REST and SOAP web services. In this section we we look at the REST web services and SOAP is covered in the last section of this chapter. As we already covered CXF earlier in this chapter, the difference between using plain Apache CXF and the camel-cxf component is the latter is Camel *first*, where as CXF is CXF *first*. Now by first we mean that its either CXF or Camel that is the primary driver behind how the wheels is spinning. So when using Camel first, the component works as any Camel component based on a consumer and producer, that are using in Camel routes.

When using camel-cxf you need to configure the CXF REST server which can be done in two kind of styles:

- *Camel configured endpoint*
- *CXF configured bean*

We will cover these two styles in the following.

CAMEL CONFIGURED ENDPOINT

A Camel configured endpoint just means you configure the CXF REST server as a regular Camel endpoint where you configure using URI notation, as shown in listing 10.5.

Listing 10.5 - Camel route using camel-cxf to define 4 REST services

```
public class OrderRoute extends RouteBuilder {
    public void configure() throws Exception {
```

```

from("cxfrs:http://localhost:8080
    ?resourceClasses=camelinaction.RestOrderService
    &bindingStyle=SimpleConsumer")
    .toD("direct:${header.operationName}");①  
②

from("direct:createOrder")③
    .bean("orderService", "createOrder");

from("direct:getOrder")④
    .bean("orderService", "getOrder(${header.id})");

from("direct:updateOrder")⑤
    .bean("orderService", "updateOrder");

from("direct:cancelOrder")⑥
    .bean("orderService", "cancelOrder(${header.id})");
}
}

```

- ① Camel configured endpoint of REST service using CXF-RS
- ② Calling the route that should perform the selected REST operation
- ③ REST service to create order
- ④ REST service to get order by id
- ⑤ REST service to update order
- ⑥ REST service to cancel order by 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 its recommended to use the SimpleConsumer binding style, that binds to the natural JAX-RS types and Camel headers. The default binding style is using the `org.apache.cxf.message.MessageContentsList` type from CXF which is a lower level type a like Camel's Exchange. The last configured options is the `resourceClass` option ① , which refers to a class or interface that declares the REST web service using JAX-RS standard.

Using class or interface for the resource class

Because CXF uses CXF-RS the REST web service is defined as a JAX-RS resource class, that lists all the REST operations. The resource class code is listing 10.6. The resource class is in fact not a class but an interface. The reason for doing so is that camel-cxf will only use the JAX-RS as a contract to setup the REST web services. At runtime the incoming requests is routed in the Camel routes instead of invoking the Java code in the resource class. And because of that its more appropriate to use an interface instead of a class with empty methods.

The REST web service has four services (as shown in listing 10.6), and which operation was called is provided in the header with name `operationName`. We use this to route the message using the dynamic to EIP pattern ② where we let each operation be a separate routes ③ ④ ⑤ ⑥ linked together using the direct component.

The actual REST web service is specified as a JAX-RS resource class which is shown in listing 10.6. In this example we use an interface ①, which is explained why in the sidebar above.

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

    @GET
    @Path("/{id}")
    Order getOrder(@PathParam("id") int orderId); ②

    @PUT
    void updateOrder(Order order); ②

    @POST
    String createOrder(Order order); ②

    @DELETE
    @Path("/{id}")
    void cancelOrder(@PathParam("id") int orderId); ②

}

```

- ① using an interface instead of a class
- ② REST operations

Each rest operation is declared using standard JAX-RS annotations ② that allows us to define the REST web services in a nice and natural way. Notice how the return types in the operations are the model types such as `Order` which returns the order details. And in the `createOrder` operation return a `String` which is the assigned order id of the just created order.

Now compare the code in listing 10.6 with listing 10.1, and notice how the return types in listing 10.1 all uses the JAX-RS `Response` type. The difference is that in listing 10.6 we can see from the interface what the REST service operations returns, such as an `Order` type in the `getOrder` operation, and a `String` type in the `createOrder` operation. Now compare this with listing 10.1, where all the REST service operations returns the same type, `Response`. When doing so we loose the ability to know exactly what the operation returns.

This can be remedied by using a 3rd 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 takes as input parameters, and what they return. Later in this chapter in section 10.3 we will learn how to use Swagger with Camel.

The source code for the book contains this example in the `chapter10/camel-cxf-rest` directory which you can try using the following Maven goal:

```
mvn test -Dtest=RestOrderServiceTest
mvn test -Dtest=SpringRestOrderServiceTest
```

You can also configure the CXF REST server using a bean style.

CXF CONFIGURED BEAN

Apache CXF was created many years ago when Spring Framework was very popular and using Spring was dominated by XML configuration. This affected CXF where 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 setup the CXF REST server using the `rsServer` element as shown in listing 10.7.

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
                  serviceClass="camelinaction.RestOrderService">
    </cxf:rsServer>

    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="cxfrs:bean:restOrderServer?bindingStyle=SimpleConsumer"/>      ②
            <toD uri="direct:${header.operationName}"/>                                ③
        </route>
        <route>
            <from uri="direct:createOrder"/>
            <bean ref="orderService" method="createOrder"/>
        </route>
        <route>
            <from uri="direct:getOrder"/>
            <bean ref="orderService" method="getOrder(${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>
```

① Setting up CXF REST server using `rsServer` element

② Camel route with the CXF configured bean

③ Calling the route that should perform the selected REST operation

The CXF REST server is configured in the `rsServer` element ① to setup 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 are using standard CXF naming. You can find many examples and details on the Apache CXF website.

The routes within the `<camelContext>` are the XML equivalent of the Java based example we covered in listing 10.5.

This example is provided as source code in the `chapter10/camel-cxf-rest` directory which you can try using the following Maven goal:

```
mvn test -Dtest=SpringRestBeanOrderServiceTest
```

The example uses XML as payload during testing, but we 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, so 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>cxfrt-rs-extension-providers</artifactId>
  <version>3.1.5</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.jaxrs</groupId>
  <artifactId>jackson-jaxrs-json-provider</artifactId>
  <version>2.7.1</version>
</dependency>
```

In the Spring or Blueprint XML file add 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 as shown:

```
<cxf:rsServer id="restOrderServer" address="http://localhost:8080"
               serviceClass="camelinaction.RestOrderService">
  <cxf:providers>
    <ref bean="jsonProvider"/>
  </cxf:providers>
</cxf:rsServer>
```

We have provided an example with the source code for the book in the `chapter10/camel-cxf-rest` directory which you can try 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. Each example has instructions how to run in the accompanying readme file.

That was a length coverage of using camel-cxf. The pros/cons is detailed in table 10.6.

Table 10.6 Pros and cons of using a 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 | The code in REST service is not executed and developers are only allowed full control what happens in Camel routes |
| CXF-RS is integrated with Camel as first class | There is no CXF in Action book |

That was a lot of converge of how to use various REST frameworks with Camel. We saw how Camel bridges to the REST services using Camel components where the REST services entry point is from a Camel route. It is as if the REST and EIP world is separated. What if you could take the REST verbs into the EIP world, so we could define REST services using the HTTP verbs in the same style as you EIPs in Camel routes. This is the topic of the next section, the Camel Rest DSL.

10.2 The Camel Rest DSL

A number of REST frameworks have emerged and gained huge popularity in recent time. For example the Ruby based web framework called Sinatra, the Node based Express framework, and from Java there is small REST and Web framework called Spark Java. What they all share in trade is to put a Rest DSL in the front of the end user making it easy to do REST development.

The story of the Rest DSL all began by conversations between James Strachan and Claus Ibsen discussing how making REST services with Apache Camel wasn't as straight-forward we would like it to be. For example the semantic 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 2 years later he writes about it in this book.

NOTE The design of Rest DSL is to make it easy and quick to define REST services at a high abstraction level, and do it *the Camel way*.

The Rest DSL debuted in end of 2014 as part of Apache Camel 2.14. However it has taken a couple of releases to further harden and improve the Rest DSL making it a great feature in the Camel toolbox.

In this section we will cover all the ins and outs of the Rest DSL, but first lets see it in action.

10.2.1 A quick example of the Rest DSL

The Camel Rest DSL was inspired by the Java Spark Rest library and so the camel-spark-rest component was the first component that is fully Rest DSL integrated. Before the Rest DSL then it is probably camel-restlet that comes closest to use the REST and HTTP verbs in the endpoint uris. If you take a moment to look at the camel-restlet example in listing 10.4, and then compare to how a similar solution is implemented with the Rest DSL as shown in listing 10.8, you should notice how the latter *speaks more* with REST and HTTP terms..

Listing 10.8 - Camel route using Rest DSL to define 4 REST services

```
public class OrderRoute extends RouteBuilder {

    public void configure() throws Exception {

        restConfiguration()
            .component("spark-rest").port(8080); ①

        rest("/orders")
            .get("{id}")
                .to("bean:orderService?method=getOrder(${header.id})") ②
            .post()
                .to("bean:orderService?method=createOrder")
            .put()
                .to("bean:orderService?method=updateOrder")
            .delete("{id}")
                .to("bean:orderService?method=cancelOrder(${header.id})");
    }
}
```

- ① Configure Rest DSL to use camel-spark-rest component
- ② Use REST verbs to define the 4 REST services with GET, POST, PUT and DELETE

When using the Rest DSL it must be configured first, where we need to tell which Rest component to use ①, and additional configurations such as the context-path and port number. In this example we have not configured any context-path so its using / as default. Then we define a set of REST services using `rest("/orders")` ② followed by each service defined using REST verbs such as get, post, put and delete. The 4 REST services will at runtime be mapped to the following URLs:

- GET /orders/{id}
- POST /orders
- PUT /orders

- DELETE /orders/{id}

Dynamic values in the context-path can be mapped using the {} syntax, which is done in the case of get and delete.

You can also use Rest DSL in the XML DSL as shown in listing 10.9.

Listing 10.9 - Camel route using Rest DSL to define 4 REST services using XML DSL

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

    <restConfiguration component="spark-rest" port="8080"/> ①

    <rest path="/orders">
        <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>
```

- ① Configure Rest DSL to use camel-spark-rest component
- ② Use REST verbs to define the 4 REST services with GET, POST, PUT and DELETE

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 using the REST verbs ② .

This example is provided as part of the source code for the book which you can try 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. Lets 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 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 is then mapped to the existing Camel DSL. By mapping to the Camel DSL it means its all essentially routes from Camel point of view.

This is better explained with an example, so lets 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 its mapped to the Camel routing DSL:

```

<route>
  <from uri="rest:get:/orders/{id}?componentName=spark-rest"/>
  <to uri="bean:orderService?method=getOrder(${header.id})"/>
  ...
</rest>
```

As you can see the route above 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"
```

We 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 how the Rest DSL maps to five individual parts in the Camel rest endpoint

1. rest - Every REST service is mapped to be using the generic Camel rest component which is provided out of the box in camel-core.
2. get - The HTTP verb such as GET, POST, PUT, etc.
3. /orders - The base url of the rest services which was configured in `<rest path="/orders">`
4. /{id} - Additional uri of the REST service, which was configured in `<get uri="{id}">`
5. componentName=spark-rest - Which Camel component to use for hosting the REST services. There can be additional parameters which is passed on from the rest configuration.

Notice how ③ and ④ is combining the base url with the uri of the REST service.

The mapping for all the 4 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
```

So we have now established how the Rest DSL is essentially a glorified syntax sugar on top of the existing DSL that essentially re-writes the REST services as small Camel routes that are just from -> to. In other words the REST services are just Camel routes which means all the

existing capabilities in Camel is being harnessed. This also means that managing these REST service becomes just as easily 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 just builds regular Camel routes which are run at runtime. So nothing to be afraid of, all what you have learned about Camel routes are useable with the Rest DSL.

If the Rest DSL is just Camel routes, what is then being used to setup a HTTP server and servicing the requests from clients calling the REST services?

10.2.3 Supported components for the Rest DSL

When using Rest DSL you need to pick one of the Rest DSL capable components that handles the task hosting the REST services. At the time of writing the following components support Rest DSL:

- camel-coap - Using Eclipse COAP for OSGi and IoT environments
- camel-netty4-http - Using the popular Netty library.
- camel-jetty - Using the Jetty HTTP server
- camel-restlet - Using the Restlet library
- camel-servlet - Using Java Servlet for Servlet or JEE servers
- camel-spark-rest - Using the Java Spark library
- camel-undertow - Using the JBoss Undertow HTTP server

You may be surprised that the list contains components that are not primary known as REST components, in fact its only camel-restlet and camel-spark-rest that 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 absent from the list is Apache CXF. The reason is that at this time of 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 there is some Apache Camel and CXF committers that 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 chose what suits best in your use case. For example you can use camel-servlet if you run your application in a servlet container such as Apache Tomcat, or JBoss Wildfly. You can also go *microservice* and run your Camel Rest applications standalone with Jetty, Restlet, Spark Rest (uses Jetty), Undertow or the popular Netty library.

So why do you need to use one of these components with the Rest DSL you may ask? The reason is we do not want to reinvent the wheel. To host REST services we need a HTTP server framework and its much better to use any of those existing, instead of building our own inside

Camel. In other words the Rest DSL has no code to deal with HTTP and REST but leave that entire up to the chosen component.

The Rest DSL makes it easy to switch between these components. All you really have to do is to change the component name in the rest configuration. However that is not entirely true as all these components have their own configuration you can use. For example you would need to do this to setup security and other advanced options.

10.2.4 Configuring Rest DSL

The Rest DSL provides configuration at the following six categories:

- common - Common options
- component - To configure options on the chose Camel component to be used
- endpoint - To configure endpoint specific options from the chose component on the endpoint that are used to create the actual consumer in the Camel route that host the REST service.
- consumer - To configure consumer specific options in the Camel route that host 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 will cover the first four categories in the following, and data format is explained in section 10.2.5 and cors headers is covered in section 10.3 as part of API documentation your rest services using swagger.

CONFIGURING COMMON OPTIONS

The common options is listed in table 10.5.

Table 10.5 Common options for the Rest DSL

| Option | Default | Description |
|-----------|---------|--|
| component | | Mandatory. The Camel component to use as the HTTP server. The chose name should be one of the supported components which is listed in section 10.2.3. |
| scheme | http | Whether to use http or https. |
| hostname | | The hostname for the HTTP server. If not specified then the hostname is resolved accordingly to the restHostNameResolver. |
| port | | The port number to use for the HTTP server. If you use the servlet component then the port number is not in use, as the servlet container controls what the actual port number would be. For example in Apache |

| | | |
|----------------------|--|---|
| | | Tomcat the default port is 8080, and in Apache Karaf / ServiceMix its 8181, etc. |
| contextPath | | Configures a base context-path to HTTP server will use. |
| restHostNameResolver | | Resolves the hostname to use, if no hostname has been explicit configured. You can specify localHostName or localIp to use either the hostname or ip address. |

All the rest configuration is configured using `restConfiguration` or `<restConfiguration>` as shown in Java:

```
restConfiguration()
    .component("spark-rest").contextPath("/myservices").port(8080);
```

... and in XML:

```
<restConfiguration component="spark-rest"
    contextPath="/myservices" port="8090"/>
```

You can try this yourself, for example lets change the example in chapter10/spark-rest to use another component such as camel-undertow. All you have to do is to change the dependency in pom.xml from camel-spark-rest to camel-undertow, and change the component name to undertow as shown below:

the changed pom.xml file

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

.. and the change in 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 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 chose Camel component. You can configure this on three levels, the component, then endpoint, and consumer. We will now take a look at how to do that.

CONFIGURING COMPONENT, ENDPOINT, AND CONSUMER OPTIONS IN REST DSL

Camel components often offers many options you can use to tweak to your needs. We have previously covered using components in chapter 6, that explains how configuration works. The

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

Rest DSL allows to configure the component directly in the rest configuration section on three different levels:

- Component level
- Endpoint level
- Consumer level

The component level is not as often used as endpoint level. As you should know so far into this book, then Camel relies heavily on endpoints which you configure as URI parameters. Therefore you will find most of the options at this level. The consumer level is seldom in use, as consumers are configured from endpoint options. Table 10.6 shows the option name to use in Rest DSL to refer to these levels.

Table 10.6 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 setup security or configure thread pool settings. At Rider Auto Parts the RESTful order application you have been working on, has a new requirement to require the clients to authenticate before they can access the services. Because Jetty supports HTTP Basic Authentication you chose 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 Rest DSL as shown as follows:

```
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 example above we lower the Jetty thread pool to use only between 1 and 8 worker threads to service incoming calls from clients.

To setup 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 a `ConstraintSecurityHandler` which we create and configure in the `JettySecurity` class as shown in listing 10.10.

Listing 10.10 - Setup Jetty Authentication using Basic Auth

```
import org.eclipse.jetty.security.ConstraintMapping;
import org.eclipse.jetty.security.ConstraintSecurityHandler;
import org.eclipse.jetty.security.HashLoginService;
import org.eclipse.jetty.security.authentication.BasicAuthenticator;
import org.eclipse.jetty.util.security.Constraint;

public class JettySecurity {

    public static ConstraintSecurityHandler createSecurityHandler() {
        Constraint constraint = new Constraint("BASIC", "customer");
①        constraint.setAuthenticate(true);

        ConstraintMapping mapping = new ConstraintMapping();
        mapping.setConstraint(constraint);
②        mapping.setPathSpec("/");

        ConstraintSecurityHandler handler = new ConstraintSecurityHandler();
        handler.addConstraintMapping(mapping);
        handler.setAuthenticator(new BasicAuthenticator());
        handler.setLoginService(new HashLoginService("RiderAutoParts",
③                "src/main/resources/users.properties"));
        return handler;
    }
}
```

- ^① Using HTTP Basic Auth
- ^② Apply to all incoming requests matching *
- ^③ Load user names from external file

To use HTTP Basic Auth we specify `BASIC` as parameter to the created `Constraint` object ^①. Then we tell Jetty to apply the security settings to the incoming requests that matches the pattern `/*` ^② which means all of them. The last part is to put all this together in a handler object that is returned. The known user and passwords is validated using a `LoginService`. Rider Auto Parts keep 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 will have to switch to a LDAP based `LoginModule` instead.

The last piece to this puzzle is to configure Rest DSL to use this security handler, which is done on the endpoint level instead of the component level. So the Rest configuration is updated as shown:

```
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 how we use the # prefix in the value to tell Camel to lookup the security handler from the Camel registry.

TIP We covered the Camel registries in chapter 4.

All there is 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:

```
<bean id="securityHandler" class="camelinaction.JettySecurity"
    factory-method="createSecurityHandler"/>
```

The source code for the book contains this example in the `chapter10/jetty-rest-security` which you can try using the following Maven goals:

```
mvn compile exec:java
```

Then from a web browser you can access the following url: `http://localhost:8080/orders/1` which should present with a login box. You can pass in jack as 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 how to run in the accompanying `readme` file.

When working with REST services then its very common to use data formats in XML or JSON formats. In the following section we will cover how to work with 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 XML/JSON contents to/from POJOs using Camel data formats. You may want to use binding if you develop POJOs that maps 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 following binding modes listed in table 10.7

Table 10.7 Binding modes supported by Rest DSL

| Mode | Description |
|----------|--|
| off | Binding is turned off. This is the default mode. |
| auto | Binding is automatic enabled if the necessary data formats 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, meaning there is no automatic binding happening for incoming and outgoing messages.

In section 10.2.2 we talked how the Rest DSL works by mapping the Rest DSL into Camel routes. When binding is enabled, then a RestBindingProcessor is injected into each Camel route as the first processor in the routing graph. 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.

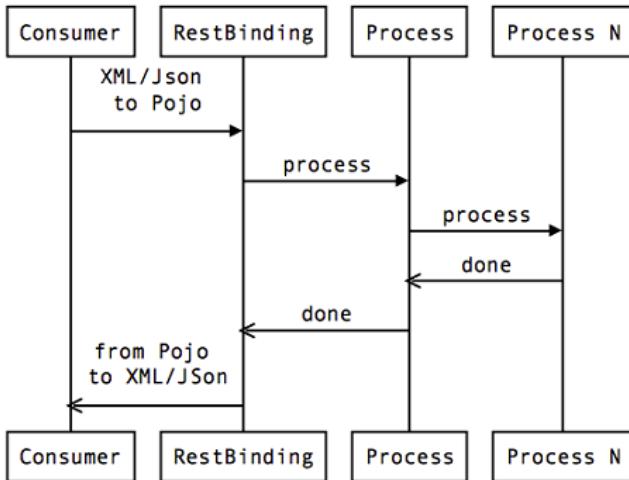


Figure 10.3 - The RestBinding sits right after the Consumer to ensure it is able first hand to bind the incoming messages from XML/Json to POJO classes, before the message is routed in the Camel route graph by the processors. When the routing is done, the outgoing message is reverse from POJO to XML/Json format right before the consumer takes over and sends the outgoing message to the client called the REST service.

The design of the Rest DSL binding is to make it easy to support the two most common data formats with REST services XML and JSON in an easy way. As a rule of thumb, you configure the binding mode to either be XML, JSON or both depending on your needs. Then add the necessary dependencies to the classpath, and you are done.

How this works as illustrated in figure 10.3 is archived from the `RestBindingProcessor` that is woven into each of the Camel routes that processes the REST services. This happens for any of the supported REST component (listed in section 10.2.3) and therefore you have binding out of the box.

Avoid double binding

Some REST libraries like Apache CXF and Restlet provides their own binding support as well. We cover using Apache CXF with JAX-RS using JSON binding in section 10.1.2. In those use-cases its not necessary to enable their binding support as well, as its provided out of the box with Camel's Rest DSL.

The `RestBindingProcessor` is responsible for handling content negotiation of the incoming and outgoing messages, which depends on a number of factors.

BINDING NEGOTIATION FOR INCOMING AND OUTGOING MESSAGES.

The `RestBindingProcessor` adheres to the following rules (in order of importance) 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, then the value of the header is used to determine if the incoming message is either in XML or JSON format.
3. If the Rest DSL has been configured with a `consumes` option, then that information is used to detect if binding from XML or JSON format is allowed.
4. If we have not yet determined if the incoming message is either in XML or JSON format, and the binding mode allows both XML and JSON, then the processor will check the message payload whether its XML content or not.
5. If the binding mode and the incoming message are incompatible then an exception is thrown.

For *outgoing* messages the ruleset is almost identical as listed:

1. If the binding mode is off, no binding takes place.
2. If the outgoing message carries a `Accept` header, then the value of the header is used to determine if the client accepts the message in either XML or JSON format.
3. If the outgoing message did not carry an `Accept` header, then the `Content-Type` header is checked whether the message body is either in XML or JSON format.
4. If the Rest DSL has been configured with a `produces` option, then that information is used to detect if binding to XML or JSON format is expected.
5. If the binding mode and the outgoing message are incompatible then 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 *aligns up* so the binding can be carried on.

Its almost time to see this in action, but first we need to know how to configure the binding.

CONFIGURING REST DSL BINDING

The binding options is listed in table 10.8.

Table 10.8 Binding options for Rest DSL

| Option | Default | Description |
|--------------------------|------------------|---|
| <code>bindingMode</code> | <code>off</code> | To enable binding where Rest DSL can automatic binding 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> , or <code>json_xml</code> . The modes are further detailed in table 10.7. |

| | | |
|------------------------|--------------|---|
| skipBindingOnErrorCode | true | Whether to use binding when returning an error as the response. This allows to build custom error messages that do not bind to json / xml etc, as success messages otherwise will do. |
| jsonDataFormat | json-jackson | 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 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 as shown below:

```
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 we have configured the binding mode to be JSON only. And to specify the JSON output should be in pretty mode, we configure this 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 and they both happen to provide an option named `prettyPrint` to turn on outputting XML/JSON in pretty mode.

When we started covering the Rest DSL in section 10.2.1, the first example was using the camel-spark-rest component, and XML was the supported content-type of the REST services. Lets see what it takes to improve this example to using JSON instead, and then there after supporting both XML and JSON.

USING JSON BINDING WITH REST DSL

First we need to add the JSON data format which is done by adding the following dependency to the Maven `pom.xml` file:

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

Then we need to enable JSON binding in the rest configuration and we also turn on pretty print mode so the JSON output from Camel are structured in a nice human readable style. Listing 10.11 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)
    .dataFormatProperty("prettyPrint", "true");

rest("/orders")
    .get("/{id}").outType(Order.class) 1
        .to("bean:orderService?method=getOrder(${header.id})")
    .post().type(Order.class) 2
        .to("bean:orderService?method=createOrder")
    .put().type(Order.class) 3
        .to("bean:orderService?method=updateOrder")
    .delete("/{id}") 4
        .to("bean:orderService?method=cancelOrder(${header.id})");
```

- ① Using JSON binding mode
- ② Turn on pretty printing for JSON output
- ③ The output type of this REST service is Order class
- ④ The input type of this REST service is Order class
- ⑤ The input type of this REST service is Order class

In the rest configuration we setup to use the camel-spark-rest component and host the REST services on port 8080. The binding mode is set to JSON ① and we want the JSON output to be pretty printed ② (uses indent and new lines) instead of outputting all the JSON data in one line only.

When you use JSON binding mode, then 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 a HTTP POST call:

```
{
    "partName": "motor",
    "amount": 1,
    "customerName": "honda"
}
```

Because we have enabled JSON binding mode, then the `RestBindingProcessor` need to transform the incoming JSON payload to a POJO class. The JSON payload do not 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)` as shown at ④ and ⑤. The REST service to return an order can likewise be marked with the outgoing type using `outType(class)` as shown at ③.

TIP Its a good practice to mark the incoming and outgoing types in the Rest DSL which is part of documenting your APIs. We will learn much more about this in section 10.4 when Swagger enters the stage.

Readers who cheer for XML should not be left out, so we have prepared listing 10.12 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">
        <dataFormatProperty key="prettyPrint" value="true"/>
    </restConfiguration>

    <rest path="/orders">
        <get uri="{id}" outType="camelaction.Order">
            <to uri="bean:orderService?method=getOrder(${header.id})"/>
        </get>
        <post type="camelaction.Order">
            <to uri="bean:orderService?method=createOrder"/>
        </post>
        <put type="camelaction.Order">
            <to uri="bean:orderService?method=updateOrder"/>
        </put>
        <delete uri="{id}">
            <to uri="bean:orderService?method=cancelOrder(${header.id})"/>
        </delete>
    </rest>
</camelContext>
```

- ① Using JSON binding mode
- ② Turn on pretty printing for JSON output
- ③ The output type of this REST service is Order class
- ④ The input type of this REST service is Order class
- ⑤ The input type of this REST service is Order class

That would be the needed changes to using JSON binding. This example is provided with the source code in chapter10/spark-rest-json directory. You can try this example using the following Maven goals:

```
mvn test -Dtest=OrderServiceTest
mvn test -Dtest=SpringOrderServiceTest
```

Now lets 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 both camel-jackson and camel-jaxb as dependency in the Maven pom.xml file. To demonstrate how easy it is to switch the rest component we 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>

```

To only changes we then have to do is to re-configure the rest configuration to use undertow and both XML and JSON binding as show below:

```

restConfiguration()
    .component("undertow").port(8080)
    .bindingMode(RestBindingMode.json_xml)
    .dataFormatProperty("prettyPrint", "true");

```

The source code for the book contains this example in the chapter10/undertow-rest-xml-json directory which you can try 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
```

Which returns the response in JSON format as shown:

```

$ 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 where you 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>
    <amount>1</amount>
    <customerName>honda</customerName>
    <id>1</id>
    <partName>motor</partName>
</order>

```

That seems easy enough? The client can now specify the output format he/she accepts and the Camel Rest DSL binding makes all that happens with little to do from the developer. Well

hold your horses .. eh Camels. When doing data transformation between XML and POJO with JAXB you need to either

- Annotate the POJO classes with a `@XmlRootElement`
- Or provide an `ObjectFactory` class

The former is the easiest but requires you to add annotations to all your POJO classes. Which you may want to do to be in more control of how XML maps to the fields in your POJO classes. Especially when you have lists and nested types. 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.

You can find examples of using annotations in many of the previous examples covered such as the example from the source code in the chapter10/spark-rest directory.

However we bring you the other side as well using the `ObjectFactory` approach which is given in the example from the source code in the chapter10/undertow-rest-xml-json directory.

TIP You can use JAXB annotations on your POJO classes which both JAXB and Jackson are able to leverage. However Jackson has its own set of annotations you can use to control the mapping between JSON and POJO.

We will 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 devoted the entire chapter five on this subject. In this section we will learn how you can 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 have amended the service to throw exceptions in case of failures as shown below:

```
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 have introduced two new exceptions. `OrderNotFoundException` is thrown if a client tries to get an order that does not exist. In that situation we want to return back a HTTP status code of 204 which means no data. The `OrderInvalidException` is thrown when a client tries to either create or update an order and the input data is invalid. In that situation we want to return

a status code 400 to the client. And finally any other kind of exceptions is regarded as an internal server error and an HTTP status code 500 should be returned.

How can you implement such a solution? Luckily you studied the error handling chapter careful and remember that Camel allows to handle exceptions using `onException` in the DSL. So you spend less than 10 minutes to add the error handling to the `OrderRoute` class that contains the Rest DSL. Listing 10.12 shows the rest configuration, error handling and rest services all together.

Listing 10.12 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
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(204))
    .setBody(constant(""));

onException(OrderInvalidException.class)          ②
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(400))
    .setBody(constant(""));

onException(Exception.class)                      ③
    .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})");
```

- ① Handle `OrderNotFoundException` and return HTTP status 204 with empty body
- ② Handle `OrderInvalidException` and return HTTP status 400 with empty body
- ③ Handle all other exceptions and return HTTP status 500 with exception message in the HTTP body

As you can see from listing 10.12 then all you did was just add three `onException` blocks ① ② ③ to catch the various exceptions and then set the appropriate HTTP status code and HTTP body.

The source code for the book contains this example in the `chapter10/undertow-rest-xml-json` directory. We have prepared a number of scenarios to demonstrate the three different exception handlers used.

For example to try to call the GET service to get an order that does not exist, you can run the following command which returns HTTP status code 204 as shown:

```
$ curl -i --header "Accept: application/json" http://localhost:8080/orders/99
HTTP/1.1 204 No Content
```

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 we have 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
```

Okay we have now covered a lot about Rest DSL, but there is more to come. In the next section we will talk about how APIs and how to document your REST services directly in the Rest DSL 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. With RESTful services using JSON, or to an less extend XML, are by far the popular choices. However this is only one part of the puzzle; how the data is structured in the payload.

To complete the jigsaw puzzle you need pieces that address technical areas such as:

- What services do a service producer offer?
- Where are the endpoints to access these services?
- What data format(s) do the services accept?
- What data format(s) can the services respond with?
- What are 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, etc.?
- What do the service do?
- What error codes can the service return?
- Is the service secured?

And besides the technical questions then as human beings we would like to known:

- What do the service do?
- What do the parameters mean?
- What do incoming and outgoing payload represent?

As you probably can tell there is a good number of other questions we could add to the bullets. 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 has service contracts built-in from the WSDL specification. This specification describes the functionality offered by the web services. For RESTful services an attempt was made with WADL to ratify as an official specification, but that did not happen. WADL and WSDL are both heavily XML savvy and primary aimed as machine readable. In practice both are unfriendly for humans to write and comprehend. Both WSDL and WADL was designed by a committee and lacked what the users wanted to use. So in the open source communities different projects came to life to build something better, 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 contains full documentation of methods, parameters, and models directly in the source code. This ensures the implementation and documentation of the RESTful services are always in sync.

Starting from 2016 the Swagger specification was renamed to OpenAPI specification and governed by founding group under the Linux Foundation. In this chapter we will 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 which 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>

In this section we will first cover how to get started with Swagger and document a JAX-RS rest service. We then see how you can provide swagger documentation directly in the Rest DSL to easily document your rest services. We finish the Swagger coverage by showing you how you can 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 it is a bit tedious to add annotations on all your rest operations, parameters and model classes.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

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>1.5.8</version>
</dependency>
```

Listing 10.13 shows the Order Service example from the Rider Auto Parts which has been documented using the Swagger annotations.

Listing 10.13 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

@Path("/orders/")
@Consumes("application/json,application/xml")
@Produces("application/json,application/xml")
@Api(value = "/orders", description = "Rider Auto Parts Order Service") ②
public class RestOrderService {

    @GET
    @Path("/{id}")
    @ApiOperation(value = "Gets an order by id", response = Order.class) ③
    public Response getOrder(@ApiParam(value = "The id of the order",
                                         required = true) @PathParam("id") int orderId) { ④
        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 = "Updates an existing order") ③
    public Response updateOrder(@ApiParam(value = "The order to update",
                                           required = true) Order order) { ④
        orderService.updateOrder(order);
        return Response.ok().build();
    }

    @POST
    @ApiOperation(value = "Creates a new order") ②
    @ApiResponses({@ApiResponse(code = 200, response = String.class,
                                 message = "The id of the created order")}) ⑤
    public Response createOrder(@ApiParam(value = "The order to create",
                                           required = true) Order order) { ④
        String id = orderService.createOrder(order);
        return Response.ok(id).build();
    }
}
```

```

    @DELETE
    @Path("/{id}")
    public Response cancelOrder(@ApiParam(value = "The order id to cancel",
                                             required = true) @PathParam("id") int orderId) {
        orderService.cancelOrder(orderId);
        return Response.ok().build();
    }
}

```

- ① Importing Swagger annotations
- ② Documenting the Rest API
- ③ Documenting each Rest operation using
- ④ Documenting each Rest input parameter
- ⑤ Documenting that the createOrder operation returns the created order id

To document JAX-RS services using Swagger you add the Swagger annotations to the JA-RS resource class. At first we import the needed Swagger annotations ① . The `@Api` annotation ② is used as high level documentation of the services in this class. In this example its the order services. Each rest operation is then documented using `@ApiOperation` ③ where we supply a description what the operation does and as well what the operation returns. Each operation that takes input parameters is documented using `@ApiParam` ④ which is co-located with the actual parameter. Swagger allows to further specify response values using `@ApiResponse` ⑤ . In this example we use this to specify that the create order operation returns the id of the created order, in the success operation. Notice how each `@ApiResponse` must map to a HTTP response code; where 200 means success.

The model classes can be annotated with `@ApiModel` as shown in listing 10.14.

Listing 10.14 Document model classes using Swagger annotations

```

import io.swagger.annotations.ApiModel;
import io.swagger.annotations.ApiModelProperty;

@XmlRootElement(name = "order")
@XmlAccessorType(XmlAccessType.FIELD)
@ApiModel(value = "order", description = "Details of the order")
public class Order {

    @XmlAttribute
    @ApiModelProperty(value = "The order id", required = true)
    private int id;

    @XmlAttribute
    @ApiModelProperty(value = "The name of the part", required = true)
    private String partName;

    @XmlAttribute
    @ApiModelProperty(value = "Number of items ordered", required = true)
    private int amount;

    @XmlAttribute
    @ApiModelProperty(value = "Name of the customer", required = true)
    private String customerName;
}

```

```
// getter/setters omitted
}
```

- ➊ Importing Swagger annotations
- ➋ Documenting the model class
- ➌ Documenting each field in the model class

Documenting model classes is easier than JAX-RS classes because you just need to document each field ➌ and the overall model ➋ as well.

After adding all the annotations to your JAX-RS and model classes, you need to integrate swagger with CXF. Doing this depends whether you use CXF in a web application, standalone, OSGi blueprint, Spring-Boot or standalone.

ADDING SWAGGER TO A CXF APPLICATION

To use Swagger with CXF you need to create an `Swagger2Feature` and configure it. This can be done in a JAX-RS application class as shown in listing 10.15.

Listing 10.15 Using JAX-RS application to setup REST service with Swagger

```
import java.util.HashSet;
import java.util.Set;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

import camelinaaction.RestOrderService;
import com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;
import org.apache.cxf.feature.LoggingFeature;
import org.apache.cxf.jaxrs.swagger.Swagger2Feature;

@ApplicationPath("/")
public class RestOrderApplication extends Application { ➊

    private final RestOrderService orderService;

    public RestOrderApplication(RestOrderService orderService) { ➋
        this.orderService = orderService;
    }

    @Override
    public Set<Object> getSingletons() { ➌
        Swagger2Feature swagger = new Swagger2Feature();
        swagger.setBasePath("/");
        swagger.setHost("localhost:9000");
        swagger.setTitle("Order Service");
        swagger.setDescription("Rider Auto Parts Order Service");
        swagger.setVersion("2.0.0");
        swagger.setContact("rider@autoparts.com");
        swagger.setPrettyPrint(true);

        Set<Object> answer = new HashSet<>(); ➍
        answer.add(orderService);
        answer.add(new JacksonJsonProvider());
        answer.add(swagger);
    }
}
```

```

        answer.add(new LoggingFeature());
        return answer;
    }
}

```

- ① JAX-RS Application class
- ② Setter to inject the RestOrderService implementation to use
- ③ Configure Swagger feature in CXF
- ④ Provide a set of features to use with CXF

JAX-RS allows you setup your JAX-RS RESTful services using your own class implementation that must extend `javax.ws.rs.core.Application` ① . In this class we provide a constructor that allows us to inject the implementation of the Rider Auto Parts order service ② . It's in the `getSingletons` method where can setup the services the RESTful application should use. To use Swagger with CXF we need to create and configure an instance of `Swagger2Feature` ③ . Then we configure the order services such as order service, Jackson for JSON support, swagger and enabling verbose logging.

To run this application standalone, we need to setup Jetty as an embedded HTTP server. And then add CXF as a servlet to the HTTP server. Listing 10.16 shows how this can be done.

Listing 10.16 Running Jetty with CXF JAX-RS application using Swagger as standalone

```

public class RestOrderServer {

    public static void main(String[] args) throws Exception {
        DummyOrderService dummy = new DummyOrderService();
        dummy.setupDummyOrders();                                ①

        RestOrderService orderService = new RestOrderService();
        orderService.setOrderService(dummy);                      ②

        RestOrderApplication app = new RestOrderApplication(orderService); ③

        Servlet servlet = new CXFNonSpringJaxrsServlet(app);
        ServletHolder holder = new ServletHolder(servlet);
        holder.setName("rider");
        holder.setForcedPath("/");
        ServletContextHandler context = new ServletContextHandler();
        context.addServlet(holder, "/*");

        Server server = new Server(9000);
        server.setHandler(context);
        server.start();                                         ⑤

        // keep that keeps the JVM running omitted
    }
}

```

- ① Use a dummy implementation as the Order Service implementation
- ② Setup the Rest Order Service
- ③ Create the JAX-RS application instance
- ④ Use CXF JAX-RS servlet without Spring
- ⑤ Setup and start Jetty embedded server on port 9000

The example is run as standalone using a plain main class. An instance of the JAX-RS application is created ③ which is then configured as a CXF JAX-RS servlet ④ . And the servlet is then added to an embedded Jetty server ⑤ which is started.

The source code for the book contains this example in the chapter10/cxf-swagger directory which you can try 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: "Gets an order by id",
    description: "",
    operationId: "getOrder",
    - consumes: [
      "application/xml",
      "application/json"
    ],
    - produces: [
      "application/xml",
      "application/json"
    ],
    - parameters: [
      - {
        name: "id",
        in: "path",
        description: "The id of the order",
        required: true,
        type: "integer",
        format: "int32"
      }
    ],
    - responses: {
      - 200: {
        description: "successful operation",
        - schema: {
          $ref: "#/definitions/order"
        }
      }
    }
  }
},
```

Figure 10.4 - Swagger API of the get order operation. Notice how the operation is detailed with a summary, what formats it consumes and produces. And details of the input parameter and as well the success response.

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 output JSON in a more human readable format. We are using JSONView extension in Google Chrome browser as shown in figure 10.4.

If you are using Rest DSL with Camel then 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. There are two things you need to do to enable Swagger with Rest DSL:

- Add camel-swagger-java as dependency
- Turn on Swagger in the Rest DSL configuration

To add camel-swagger-java as dependency you can add the following to your Maven pom.xml file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-swagger-java</artifactId>
    <version>2.17.0</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 source code for the book contains this example in the chapter10/undertow-swagger directory which you can run using the following Maven goal:

```
mvn compile exec:java
```

then the Swagger API is available in the following url:

```
http://localhost:8080/api-doc
```

TIP You can specify whether the Swagger API should output using JSON or Yaml. Use `http://localhost:8080/api-doc/swagger.json` for JSON and `http://localhost:8080/api-doc/swagger.yaml` for YAML.

So 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 that are defined in the Rest DSL. When `apiContextPath` is configured, then the Swagger API is turned on. Under the hood Camel creates a route that routes from the `api context-path` to a `rest-api` endpoint. In Camel route lingo this would be:

```
from("undertow:http://localhost:8080/api-doc")
    .to("rest-api://api-doc");
```

The `rest-api` is a component from `camel-core` that seamless integrates with Swagger in the `camel-swagger-java` module. When Camel startup the `rest-api` endpoint will then discover that `camel-swagger-java` is available in the classpath and then trigger the Swagger integration. What happens next is that the model of the rest services from Rest DSL is parsed by a Swagger model reader from the `camel-swagger-java` module. The reader then transforms the Rest DSL into Swagger API model. When a request to the Swagger API service then 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. However they can be used to documenting your model objects. Besides using the Swagger annotations you document your services directly using the Rest DSL.

10.3.3 Documenting Rest DSL services

In the section introduction we discussed that RESTful services benefits from being documented in a way that allows both humans and computers to understand all the capabilities of the services without having to lookup information from external resources such as offsite documentation or source code.

HATEOAS

HATEOAS (Hypermedia As The Engine Of Application State) is an extension to RESTful principles that the services should be hypermedia driven. This principle goes the extra mile where a client needs no prior knowledge how to interact with a service, besides the basic principles of hypermedia. Trying to explain this in other words would be like humans can interact with any website using a web browser and follow hyperlinks. That is the same principle for the HATEOAS where links are returned in responses to clients, that can follow these links and so on. The Rest DSL does not yet support HATEOAS.

The Rest DSL allows to document the Rest services directly in the DSL to document operations, parameters, response codes and types, and the likes.

Listing 10.13 shows how the Rider Auto Parts order service application has been documented:

Listing 10.13 Document services using Rest DSL

```
rest("/orders")
    .description("Order services")  
①  

    .get("{id}").outType(Order.class)
        .description("Get order by id")
        .param().name("id").description("The order id").endParam()②
        .to("bean:orderService?method=getOrder(${header.id})")  

    .post().type(Order.class).outType(String.class)
        .description("Create a new order")  
①
        .responseMessage()
            .code(200).message("The created order id")  
③
        .endResponseMessage()
        .to("bean:orderService?method=createOrder")  

    .put().type(Order.class)
        .description("The order to update")  
①
        .to("bean:orderService?method=updateOrder")  

    .delete("{id}")
        .description("The order to cancel")
        .param().name("id").description("The order id").endParam()②
        .to("bean:orderService?method=cancelOrder(${header.id})");
```

- ① Description of the REST service and operations
- ② Input parameter documentation
- ③ Response message documentation

You use `description`, `param` and `responseMessage` to document your Rest DSLs as shown in listing 10.13. The `description` ① is used to provide a summary of the rest service and operations. The `param` ② is used for documenting input parameters. And `responseMessage` ③ is used to document responses.

In listing 10.13 there are two services which have input parameters ② :

```
.param().name("id").description("The order id").endParam()
```

The parameter is named `id`, and its 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 .

The accompanying source code of the book contains this example using Java DSL in the `chapter10/undertow-swagger` directory which you can try using:

```
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/servertl-swagger-xml directory. This example is deployable as a web application, so you can deploy the .war artifact that is built into Apache Tomcat or WildFly servers. You can also run the example directly from Maven using

```
mvn compile jetty:run
```

You also need to document what input and output your rest services accepts and returns.

10.3.4 Documenting input, output and error codes

In this section we will cover how you configure and document what input data and parameters your rest services accepts. Then we move on how to document what your rest services can send as output responses. And we end the section covering how do document failures.

A RESTful service can have a number of input parameters in different form and shape that total up to five different types:

- *body* - The HTTP body
- *formData* - A form in the HTTP body
- *header* - A HTTP header
- *path* - A context-path segment
- *query* - A HTTP query parameter

The most common parameter types in use are body, path and query. In the previous example we were using two parameter types: body and path.

THE INPUT PATH PARAMETER TYPE

The following snippet is the service 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 only in use when the service uses `{ }` in the url. As you can see from the snippet the url is configured as `{id}` that will automatically let Rest DSL define a API documentation for this parameter.

You can explicitly declare the parameter type in the Rest DSL and configured additional details as shown:

```
.param().name("id").description("The order id").endParam()
```

You can also specify what the expected data type is. For example to document 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 only want to do this 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 automatic defines that as a body parameter type. Because the type is a POJO then the data type of the parameter is a schema type. In the generate Swagger API documentation the schemas are listed in the bottom.

Figure 10.5 shows how the Swagger API documentation maps the body parameter type to the order definition.

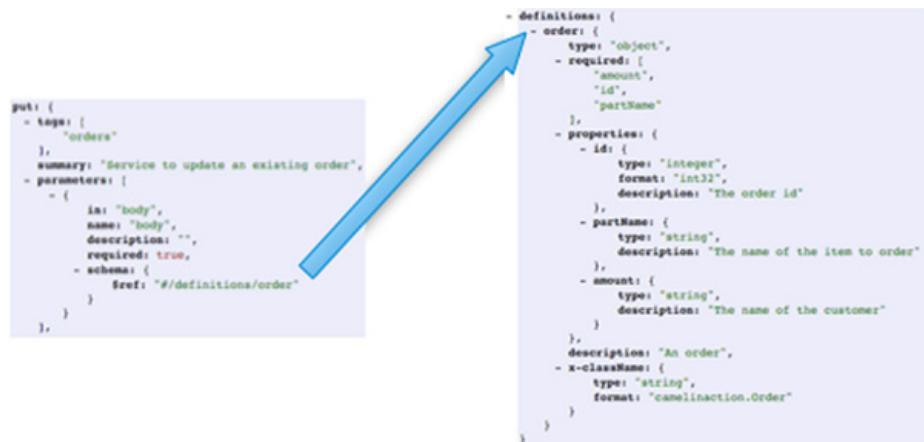


Figure 10.5 - The figure on the left hand side shows the PUT operation with a body input parameter that refers to a schema definitions/order. The schema is defined in the bottom of the Swagger API documentation and shown in the figure on the right hand side.

OTHER INPUT PARAMETER TYPES

All the parameter types is configured in the same style, for example to specify a query parameter named priority you do 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 have covered the input parameter types. But there is 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 - A HTTP header

Lets 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 we are using `outType(Order.class)` that causes Rest DSL to automatic define the response message as an Order type. Because the out type is a POJO then the data type of the response is a schema type. Figure 10.6 shows how the generated Swagger API documentation for this rest service and how it refers to the order schema.



Figure 10.6 - The figure on the left hand side shows the GET operation with a successful (code 200) response body that refers to a schema definitions/order. The schema is defined in the bottom of the Swagger API documentation and shown in the figure on the right hand side.

You can also explicitly define response messages where you can humanly describe the response message as highlighted below:

```
.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 where 200 is a successful call. In a short while we will show you how you document failures.

When a rest service returns a response then 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 responses. If you do so then you can document these headers in the Rest DSL as highlighted below:

```
.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 different status codes as listed in table 10.9.

Table 10.9 - The four possible HTTP status codes the Rider Auto Parts order application can return

| Code | Exception | Description |
|------|------------------------|---|
| 200 | None | The request was processed successfully. |
| 204 | OrderNotFoundException | A client attempted to get status of a non existing order. Therefore response code 204 is returned to indicate no data. |
| 400 | OrderInvalidException | A client attempted to create or update an order with invalid input data. Therefore a client error code 400 is returned to indicate invalid input. |
| 500 | 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(204).message("Order not found").endResponseMessage()
.responseMessage()
.code(500).message("Server error").endResponseMessage()
.to("bean:orderService?method=getOrder(${header.id})")
```

And the 2nd 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>
```

So how do you handle those thrown exceptions in the Rest DSL? Well this is done using regular Camel routes where you use Camel's error handler such as `onException`. The following snippets shows how we handle the generic exception and transform that into a 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 We cover all about error handling in the following chapter. So do not worry if you do not completely understand.

TIP We cover all about error handling in the following chapter. So don't worry if you do not completely understand all details of `onException`. All that is extensively covered in the next chapter.

We encourage you to try the example that is accompanying 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 has a `readme` file with further instructions.

When using API documentation there is a number of configuration options you may need to use.

10.3.5 Configuring API documentation

The Rest DSL provides a number of configuration options for configuring API documentation as listed in table 10.10.

Table 10.10 API Documentation options for Rest DSL

| apiComponent | swagger | The name of the Camel component to use as the REST API (such as swagger) |
|---------------------|---------|--|
| apiContextPath | | Configures a base context-path the Rest API server will use. Important: You must specify a value for this option to enable API documentation. |
| apiContextRouteId | | To specific the name of the route that the Rest API server creates for hosting the API documentation. |
| apiContextIdPattern | | Sets a pattern to only expose Rest APIs from REST services that are hosted within CamelContext's matching the pattern. You use this when running multiple Camel REST application in the same JVM and you want to filter which CamelContext's are exposed as Rest APIs. The pattern <code>#name#</code> refers to the CamelContext name, to match on the current CamelContext only. |
| apiContextListing | false | Sets whether listing of all available CamelContext's with REST services in the JVM is enabled. If enabled it allows to discover these contexts, if false then only the current CamelContext is in use. |

| | | |
|------------|-------|---|
| enableCORS | false | Whether to enable CORS headers in the HTTP responses. |
|------------|-------|---|

You will find yourself often in need to only use the `apiContextPath` option as its the one that enables API documentation. You will not use the first option as currently Swagger is the only library that is integrated with Camel to service RESTful API documentation.

The 3rd, 4th and 5th option are all related to filter out which CamelContext's and routes to include in the API documentation. The last option is for enabling CORS which we will cover at the end of this section.

ENABLING API DOCUMENTATION

To use Swagger API you need to enable this in the Rest DSL configuration. To enable Swagger API you configure a context-path which is used as base path to service the API documentation.

The following snippet highlights how to configure this in Java and XML DSL:

```
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 there is a set of general information, such as version and contact information, which you likely want to provide in the documentation. The information is configured using `apiProperty` as shown below:

```
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 that allows customers 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 that allows customers 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 which you can find more details at the following url: <http://swagger.io/specification/#infoObject>.

FILTERING CAMELCONTEXT AND ROUTES IN API DOCUMENTATION

You may have some rest services which you do not want to be included in the public Swagger API documentation. The easiest way to disable a rest service is to set the `apiDoc` attribute to `false`, as shown:

```
.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 only find useable 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 which can discover and service API documentation for all the Camel applications running in the same JVM. This allows 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`, then the root path returns all the ids of the Camel applications that has 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 output above there is only one Camel application in the JVM that has RESTful services. The name is `camel-1`, which means the API documentation from within that Camel application is available in the following url:

```
$ curl http://localhost:8080/chapter10-servlet-swagger-xml/rest/api-doc/camel-1
{
  "swagger" : "2.0",
  "info" : {
    "description" : "Order Service that allows customers to submit orders and query status",
    "version" : "2.0.0",
    "title" : "Rider Auto Parts Order Services",
    "contact" : {
      "name" : "Rider Auto Parts"
    }
  }
}
```

```

    }
}
...

```

Swagger allows to visualize the API documentation using 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 Swagger web console

Web browsers have over the years become more secure and cannot access resources that are outside the current domain. So if a user visits a web page <http://rider.com> then that website can freely load resources from the `rider.com` domain. But if the webpage attempts to load resources from outside that domain, then the web browser would disallow that.

However it is not that simple as some resources are always allowed such as CSS styles, images, scripts, etc., but advanced requests such as POST, PUT, DELETE etc. are not allowed.

CORS (Cross Origin Resource Sharing) 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 where the client specifies what resource it wants access to, and the server then responds with a set of HTTP headers that tells the client what is 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? For example if you build web applications using JavaScript technology that call RESTful services on remote servers.

Its easy to enable CORS when using Rest DSL which is simply done in the rest configuration as shown:

```

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 then the following CORS headers as listed in table 10.11.

Table 10.11 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 values is not what you need then you can customize the CORS headers in the rest configuration as shown:

```
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 that allows customers 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 that allows customers 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 then only clients from the domain `rider.com` is allowed to access the RESTful services using CORS. The Max-Age option is set to 5 minutes which the clients is allowed to cache a pre-flight requests to a resource before the client must re-new and call a new pre-flight request.

Lets put CORS to use and use the Swagger UI web application.

EMBEDDING SWAGGER UI WEB CONSOLE

The Swagger API documentation can be visualized using Swagger UI. This web console also allows 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 more easily host or embed the console on application servers or locally.

The source code for this book contains an example where the web console is embedded together in a Camel application. The example is located in the chapter10/swagger-ui directory which you can start 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 present 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 and type in 1 as order id, and click the try it button.

Now lets try to browse Swagger API documentation from a remote service. We have another example in the source code in chapter10/spark-rest-ping directory. Now start this example using

```
mvn compile exec:java
```

Which starts a ping rest service that you can access with

```
http://localhost:9090/ping
```

And the Swagger API for the ping service is available at:

```
http://localhost:9090/api-doc
```

You can then from the Swagger UI view the ping service by typing in the top of the web page the url for the ping API documentation, and click the explore button, as shown in figure 10.7.

Figure 10.7 Swagger UI browsing the remote ping rest API documentation. In the top of the window you type the URL to the Swagger API you want to browse and click Explore button. Then you can Expand Operations and try calling the services. In the figure we have called the ping service and received a pong reply in the Response Body section.

The Swagger UI is only able to browse and call the remote rest services because we have 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: Fri, 25 Mar 2016 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(9.2.15.v20160210)

{ "reply": "pong" }
```

Whoa that was a lot of content to cover about RESTful services. Who would have thought there is so much to learn and master using such a popular and modern means of communication.

Now lets step back 10 years and talk about what some people may regard as the predecessor to RESTful web services which is the famous SOAP based web services. If you are lucky and do not have to use SOAP web services anymore then you can skip the section and go straight to the chapter summary.

10.4 Web services

A while back you would be hard pressed to find any existing enterprise project that doesn't use web services of some sort. They're an extremely useful integration technology for distributed applications. Web services are often associated with service-oriented architecture (SOA), where 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 what operations you can call on a web service and what the input and output types are, among other things. Messages are typically XML, formatted to comply with the Simple Object Access Protocol (SOAP) schema. In addition, these messages are typically sent over HTTP. As illustrated in figure 10.8, web services allow you to write Java code and make that Java code callable over the Internet, which was pretty neat when web services became popular a decade ago!

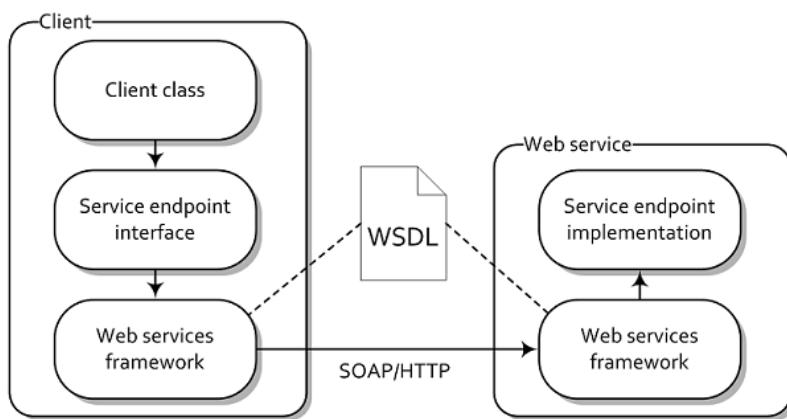


Figure 10.8 A client invokes a remote web service over HTTP. To the client, it looks as if it's calling a Java method on the service endpoint interface (SEI). Under the hood, this method invocation passes through the web services framework, across a network, and finally calls into the service endpoint implementation on the remote server.

For accessing and publishing web services, Camel uses Apache CXF (<http://cxf.apache.org>). CXF is a web services framework that supports many web services standards, most of which we won't discuss here. We'll mainly be focusing on developing web services using the Java API

for XML Web Services (JAX-WS) specification. JAX-WS defines annotations that allow you to tell a tool like CXF how your POJO should be represented on the web.

TIP Camel also provides the `camel-spring-ws` component that uses Spring's Web Services framework. However Apache CXF is the most used and best supported web service component that Camel offers.

We'll be covering two types of web services development with CXF in this section:

- *Contract-first development*—Recall that WSDLs define what operations and types a web service provides. This is often referred to as a web services contract, and in order to communicate with a web service, you must satisfy the contract. Contract-first development means that you start out by writing a WSDL file (either by hand or with the help of tooling), and then generating stub Java class implementations from the WSDL file by using a tool like CXF.
- *Code-first development*—The other way to develop web services is by starting out with a Java class and then letting the web service framework handle the job of generating a WSDL contract for you. This is by far the easiest mode of development, but it also means that the tool (CXF in this case) is in control of what the contract will be. When you want to fine-tune your WSDL file, it may be better to go the contract-first route.

To show these concepts in action, let's going back to Rider Auto Parts, where they need a new piece of functionality implemented. In chapter 2 (figure 2.2) you saw how 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 backend order processing systems needed to be a web service.

Before you jump into creating this web service, let's take a moment to go over how CXF can be configured within Camel.

10.4.1 Configuring CXF

There are two main ways to configure a CXF component URI: by referencing a bean containing the configuration or by configuring it within the URI. The former is the *code-first development* and the latter is *contract-first development*.

CONFIGURING USING URI OPTIONS

When configuring CXF using only URI options, a CXF endpoint URI looks like this,

```
cxft://anAddress[?options]
```

where `anAddress` is a URL like `http://rider.com:9999/order`, and `options` are appended as usual from the possible options in table 10.12.

Table 10.12 Common URI options used to configure the CXF component

| Option | Description |
|--------------|--|
| serviceClass | <p>Specifies the name of the service endpoint interface (SEI). Typically this interface will have JAX-WS annotations. The SEI is required if the CXF data format mode is POJO. If you already have an instance of a concrete class, you can reference it using the #beanName style.</p> <p>Configuring CXF using a serviceClass bean is the most common way of using CXF with Camel.</p> |
| serviceName | <p>Specifies the service to use. The format is a qualified name (QName) that has a namespace and name like { http://order.camelinaction}OrderEndpointService .</p> <p>Note that if there is only one service in a WSDL, Camel will choose this as the default service. If there is more than one service defined, you need to set the serviceName property.</p> |
| portName | Specifies the port to use. The format is a qualified name (QName) that has a namespace and name like { http://order.camelinaction }OrderService. |
| dataFormat | Sets the data format type that CXF uses for its messages. The possible values are POJO, PAYLOAD, and MESSAGE. We'll only be covering POJO mode in this chapter; you can find more information on the other two modes on the CXF component page in the Camel online documentation: http://camel.apache.org/cxf.html . |
| wsdlUrl | Specifies the location of the WSDL contract file. If using serviceClass then the WSDL is automatic generated by CXF and this option is not applicable. However if no serviceClass has been configured you must configure this option to refer to an existing WSDL file to use. |

CONFIGURING USING A CXF ENDPOINT BEAN

When using a CXF endpoint bean in Spring, you have much more power than by configuring CXF via URI options. In the CXF endpoint bean, you can configure things like CXF interceptors, JAX-WS handlers, and the CXF bus. The URI for configuring the CXF component looks like this:

```
cxf:bean:beanName
```

The `beanName` name specifies the ID of the CXF endpoint bean defined in your Spring XML file. This bean supports the URI options listed in table 10.12 as well as an `address` attribute that tells Camel what address to use for the web service.

Listing 10.17 shows how a CXF endpoint bean is configured.

Listing 10.17 The CXF endpoint bean format

```
<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">
```

1

```

<cxf:cxfEndpoint
    id="orderEndpoint"
    address="http://localhost:9000/order/"
    serviceClass="camelinaction.order.OrderEndpoint"/>

```

②

```
</beans>
```

- ① Declares the Camel CXF namespace
- ② Sets bean ID, address, and SEI

The CXF endpoint is configured using the Camel CXF namespace ① and therefore you must prefix the bean with `cxf` which becomes `<cxf:cxfEndpoint>` in the XML file. The CXF endpoint is configured with a `serviceClass` and the URL to use as endpoint for clients to access the web service ②.

After configuring an endpoint as shown in listing 10.17, you can use it in a producer or consumer using the following URI:

```
cxf:bean:orderEndpoint
```

There is a notable difference when using producers versus consumers.

PRODUCERS VERSUS CONSUMERS

In the context of a web service, a *producer* in Camel calls a remote web service. This web service could be defined by Camel or by some other web framework.

To invoke a web service in Camel, you use the familiar `to` Java DSL method:

```
.to("cxf:bean:orderEndpoint");
```

Consumers are a little more interesting, as they expose an entire route to the world as a web service. This is a powerful concept. A Camel route could be a complex process, with many branches and processing nodes, but the caller will only see it as a web service with input parameters and a reply.

Say you start out with a route that consists of several steps, like this:

```
from("jms:myQueue").
    .to("complex step 1").
    ...
    .to("complex step N");
```

To expose this route to the web, you can add a CXF endpoint to the beginning:

```
from("cxf:bean:myCXFEndpointBean").
    .to("complex step 1").
    ...
    .to("complex step N");
```

Now, when the web service configured by `myCXFEndpointBean` is called, the whole route will be invoked.

TIP If you're coming from a background in SOA or have used web services before, you may be scratching your head about consumers in Camel. In the web services world, a consumer is typically a client that calls a remote service. In Camel, a consumer is a server, so the definition is reversed!

MAVEN DEPENDENCIES

In order to use the CXF component, you'll have to add some dependencies. First, you need to depend on the camel-cxf module:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cxf</artifactId>
  <version>2.17.0</version>
</dependency>
```

That will get you most of the way to a usable CXF component, but you also need to add a module for the CXF transport you're using. In most cases, this will be HTTP, so you'll need to add another dependency to your POM:

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http-jetty</artifactId>
  <version>3.1.5</version>
</dependency>
```

CXF supports several other transports as well, and you can find more information about them on the CXF website at <http://cxf.apache.org/docs/transports.html>.

Now that you have a sense of the configuration details, let's take a hands-on look at how to develop web services with Camel.

10.4.2 Using a contract-first approach

In contract-first development, you start by creating a WSDL document and then getting a web service tool to generate the necessary Java code. This process is illustrated in figure 10.9.

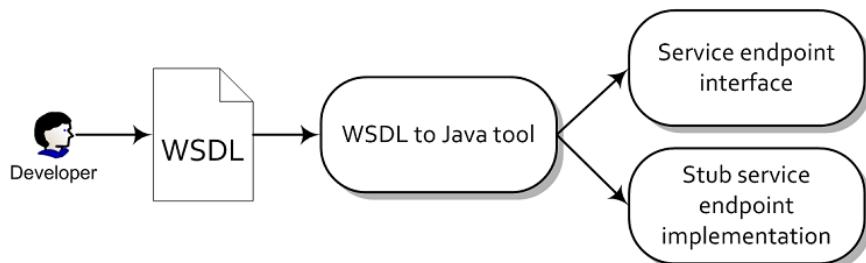


Figure 10.9 In contract-first web service development, you start out by creating a WSDL document and letting a tool generate the required source interfaces and stubs.

Creating the WSDL contract for a particular web service is a non-trivial task. It's often best to think about what methods, types, and parameters you'll need before starting.

In this case, to place an order at Rider Auto Parts with a web service, you need a single method call named `order`. This method will accept part name, amount, and customer name parameters. When the method is complete, it will return a result code to the client. The web service should be exposed on the server's `http://localhost:9000/order` address.

The WSDL for this web service is shown in listing 10.18.

Listing 10.18 The WSDL for an order service

```

<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://order.camelinaction"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://order.camelinaction">
    ①
    <wsdl:types>
        <xss:schema targetNamespace="http://order.camelinaction">
            <xss:element type="xss:string" name="partName" />
            <xss:element type="xss:int" name="amount" />
            <xss:element type="xss:string" name="customerName" />
            <xss:element type="xss:string" name="resultCode" />
        </xss:schema>
    </wsdl:types>
    <wsdl:message name="purchaseOrder">
        ②
        <wsdl:part name="partName" element="tns:partName" />
        <wsdl:part name="amount" element="tns:amount"/>
        <wsdl:part name="customerName" element="tns:customerName"/>
    </wsdl:message>
    <wsdl:message name="orderResult">
        <wsdl:part name="resultCode" element="tns:resultCode" />
    </wsdl:message>
    <wsdl:portType name="OrderEndpoint">
        ③
        <wsdl:operation name="order">
            <wsdl:input message="tns:purchaseOrder" />
            <wsdl:output message="tns:orderResult" />
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="OrderBinding"
        type="tns:OrderEndpoint">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="order">
            <soap:operation
                soapAction="http://order.camelinaction/Order" style="document" />
                ④
                <wsdl:input>
                    <soap:body parts="in" use="literal" />
                </wsdl:input>
                <wsdl:output>
                    <soap:body parts="out" use="literal" />
                </wsdl:output>
            </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="OrderEndpointService">
            ⑤
            <wsdl:port name="OrderService" binding="tns:OrderBinding">
                <soap:address location="http://localhost:9000/order" />
            </wsdl:port>
        </wsdl:service>
    </wsdl:bindings>

```

```

</wsdl:port>
</wsdl:service>
</wsdl:definitions>

① Defines input and output parameters
② Defines messages
③ Defines interface to call
④ Uses SOAP encoding over HTTP transport
⑤ Exposes service using interface

```

As you can see in listing 10.18, a WSDL contract is quite a mouthful! Writing this kind of document from scratch would be pretty hard to get right. Typically, a good way to start one of these is to use a wizard or GUI tooling. For instance, in Eclipse you can use the File > New > Other > Web Services > WSDL wizard to generate a skeleton WSDL file based on several options. Tweaking this skeleton file is much easier than starting from scratch.

CXF also has command-line tools to help you create a WSDL contract properly. Once you have a `portType` element defined, you can pass the WSDL fragment through CXF's `wsdl2xml` tool (<http://cxf.apache.org/docs/wsdl-to-xml.html>), which will add a binding element for you. When the binding element is defined, the `wsdl2service` tool (<http://cxf.apache.org/docs/wsdl-to-service.html>) can then generate a service element for you.

There are five main elements specified in the WSDL file shown in listing 10.18, and all WSDL files follow this same basic structure:

- `types`—Data types used by the web service
- `message`—Messages used by the web service
- `portType`—Interface name and operation performed by the web service
- `binding`—Transport type and message encoding used by the web service
- `service`—Web service definition, which specifies the binding to use as well as the network address to use

You first define what parameters the web service will be passing around ① . This configuration is done using the XML schema, which may make it a bit more familiar to you. You specify a name and a type for each parameter.

The next section of listing 10.18 defines the messages used by the web service ② . These messages allow you to assign parameters to use as input versus output.

You then define the `portType` ③ , which is the interface that you'll be exposing over the web. On this interface, you define a single operation (method) that takes a `purchaseOrder` message as input and returns an `orderResult` message.

The `binding` section ④ then specifies the use of the HTTP transport for the messages and that the messages should be encoded over the wire using *document literal* style. Document literal means that the SOAP message body will be an XML document. The format of this XML document is specified using the XML schema.

Finally, the `service` section ⑤ exposes a port using a specific binding on an address. There can be more than one port listed here. In this example, you use the port and binding definitions from before and set the web service address to `http://localhost:9000/order`.

The next step in contract-first web service development is taking the WSDL and generating Java code that implements it. CXF provides the `wsdl2java` tool (<http://cxf.apache.org/docs/wsdl-to-java.html>) to do this for you. Listing 10.19 shows how this tool can be used from a Maven POM file.

Listing 10.19 Using CXF's wsdl2java tool

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>3.1.5</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>
          ${basedir}/target/generated/src/main/java
        </sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>
              ${basedir}/src/main/resources/wsdl/order.wsdl
            </wsdl>
            <extraargs>
              <extraarg>-impl</extraarg>
            </extraargs>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

- ① Location of generated source
- ② Location of input WSDL

The tool accepts your WSDL file ② and an output location for the generated source ① . To run this tool for yourself, change to the `chapter10/contract-first` directory and run the following command:

```
mvn generate-sources
```

After this completes, you can look in the output directory and see that there are four files generated:

```
ObjectFactory.java
```

```
OrderEndpointImpl.java
OrderEndpoint.java
OrderEndpointService.java
```

These files implement a stubbed-out version of the order web service. If you were not using Camel, you would write your business logic in the `OrderEndpointImpl` file that was generated.

To use this web service in Camel, you need to define a CXF endpoint. Listing 10.20 shows how to do this in Spring.

Listing 10.20 CXF endpoint configuration in camel-cxf.xml

```
<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">

    <cxf:cxfEndpoint id="orderEndpoint"
                      address="http://localhost:9000/order/"
                      serviceClass="camelinaction.order.OrderEndpoint" ①
                      wsdlURL="wsdl/order.wsdl"/>
</beans>
```

① Defines endpoint to be used from Camel

This endpoint configures CXF under the hood to use the web service located at `http://localhost:9000/order` and using the `camelinaction.order.OrderEndpoint` interface. Because there is only one service defined in `order.wsdl`, CXF will choose that automatically. If there were more than one service, you would need to set the `serviceName` and `endpointName` attributes on the endpoint bean. The `serviceName` is the name of the WSDL service element, and `endpointName` is the name of the port element.

You can browse a web service's WSDL yourself by appending `?wsdl` to any web service URL in your browser. For this address, that would be `http://localhost:9000/order?wsdl`. This WSDL is the same as the file provided in the `wsdlURL` attribute of the endpoint bean.

ADDING A WEB SERVICE TO YOUR ROUTE

With all that set up, you're ready to start using the order web service within a Camel route. Listing 10.21 shows a route using the web service.

Listing 10.21 Web-enabled route configuration

```
<import
      resource="classpath:META-INF/spring/camel-cxf.xml" /> ①

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="cxf:bean:orderEndpoint" /> ②
    </route>
</camelContext>
```

```

<to uri="seda:incomingOrders" />
<transform>
    <constant>OK</constant>
</transform>
</route>
</camelContext>

```

3

- ① Imports CXF endpoint bean
- ② Exposes route as web service
- ③ Returns OK reply to caller

Because you defined your CXF web service as an endpoint bean in listing 10.20, you just had to import the bean configuration ① and refer to the bean id to set up the consumer ② . Recall that setting up a CXF consumer effectively turns the entire route into a web service, so once the order data is sent to an internal queue for processing, the output produced by the route ③ is returned to the caller of the web service ② .

How does one call this web service? Well, you could use pure CXF or another web services framework compatible with CXF. In this case, you'll use Camel. You used an endpoint bean to configure CXF earlier, so you can use that to send to the web service as well.

You first need to prepare the parameters to be passed into the web service:

```

List<Object> params = new ArrayList<>();
params.add("motor");
params.add(1);
params.add("honda");

```

Recall that, in the WSDL, you specified that the web service accepted parameters for part name, number of parts, and customer name.

Next, you can use a `ProducerTemplate` to send a message to the web service:

```

String reply = template
    .requestBody("cxf:bean:orderEndpoint", params, String.class);
assertEquals("OK", reply);

```

To try this out for yourself, change to the `chapter10/contract-first` directory, and run the following Maven command:

```
mvn test
```

This will run the `wsdl2java` tool to generate the code from the WSDL, and then run a test that loads up the web-enabled route and calls it using a `ProducerTemplate` and asserts the reply from the web-service is OK.

SELECTING THE OPERATION WHEN INVOKING WEB SERVICES

If you call a web service with multiple operations, you need to specify which operation Camel should invoke. You can do this by setting the `operationName` header on the message before sending it to a CXF endpoint. Here is an example:

```
<route>
```

```

<from uri="direct:startOrder" />
<setHeader headerName="operationName">
    <constant>order</constant>
</setHeader>
<to uri="cxf:bean:orderEndpoint"/>
</route>

```

In this case, you're invoking the `orderEndpoint`, which only has one operation, but this demonstrates how you can use this header. The header is set to the operation name `order`, which you can find in the WSDL under the `wsdl:operation` element.

Think contract-first development is hard? Well, some developers do, even though it gives you complete control over your web service contract—an important detail. Next we'll look at how to develop web services using a code-first approach.

10.4.3 Using a code-first approach

Code-first web services development is often touted as a much easier alternative to contract-first development. In code-first development, you start out with Java that's annotated with JAX-WS annotations, and then you get the web services framework to generate the underlying WSDL contract for you. This process is illustrated in figure 10.10.

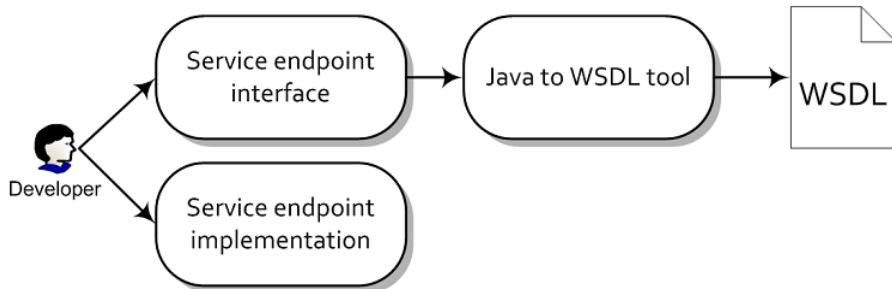


Figure 10.10 In code-first web services development, you start out by coding the service interface and implementation and then using a tool to generate the required WSDL.

To see how this is possible with Camel, let's try implementing the solution presented in the previous section in a code-first manner.

You start much as you do with contract-first development—you need to think about what methods, types, and parameters you need. Because you're starting with Java, you'll need an interface to represent the web service:

```

@WebService
public interface OrderEndpoint {
    String order(String partName, int amount, String customerName);
}

```

The JAX-WS `javax.jws.WebService` annotation will tell CXF that this interface is a web service. There are many annotations that allow you to fine-tune the generated WSDL, but for many cases the defaults work quite well.

In order to use this interface as a web service, you don't have to change any of your Camel configuration from the previous section. Yes, you read that correctly. Your CXF endpoint bean is still defined as follows:

```
<cxf:cxfEndpoint id="orderEndpoint"
    address="http://localhost:9000/order/"
    serviceClass="camelinaction.order.OrderEndpoint"/>
```

To use this bean in a Camel route, you can reference it as before:

```
<from uri="cxf:bean:orderEndpoint"/>
```

To try this out for yourself, change to the `chapter10/code-first` directory and run the following Maven command:

```
mvn test
```

This will run a test that loads up the web-enabled route and calls it using a `ProducerTemplate`. The Java to WSDL process happens automatically in the background.

Certainly, the code-first approach allows you to implement web services quickly. But it's good to understand what is happening under the hood of a web service, because it's a complex technology.

That's it folks. That is all we wanted to say about Web Services. Hopefully you would encounter them less and less in your integration projects and use more modern technologies like REST services and what comes around the corner in the future. Some folks are whispering in the corners about event sourcing. But we have yet that to see, and it would also be topic for a chapter in a future book.

10.5 Summary and best practices

In this chapter we look at how to build RESTful services with Camel, using the wide range of different choices Camel offers on the table. In fact we started without Camel and build a pure JAX-RS service using CXF and then introduce Camel in various stages. We went through all the other rest components that are at your disposal, and provided a short summary of the pros and cons of each choice.

Rest services is becoming a top choice for providing and also 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-fact standard for 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 to browse and access remote rest services.

The last section of this chapter is a step back in time to cover old school SOAP based web services. You saw how to build contract and code first web services with CXF and Camel.

We have listed the bullets into two groups as the take away from this chapter. The first group is about RESTful services, and the latter about Web Services.

The take away for RESTful services:

- *JAX-RS is 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 how to handle the REST services. Apache CXF integrates the JAX-RS specification and is a good RESTful framework.
- *Java code or Camel route.* Apache Camel allows to expose Camel routes as if they are RESTful services. You need to decide whether you want the full coding power that JAX-RS resource class allows, or to go straight to Camel routes. When you are in need of both, then you can use the hybrid mode with a JAX-RS resource class and call Camel routes using a 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 to REST to quickly understand and develop REST services that integrates well with the philosophy and principles of Apache Camel.
- *Use the right REST components.* The Rest DSL allows to use many different 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 JEE 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 automatic provide API documentation using Swagger at runtime.
- *Govern and manage your APIs.* There was no room in this chapter to touch this topic, but we wanted to share our views that API management is becoming important when there are many more APIs to integrate. External projects such as Apiman providers a very powerful API governance and management platform. Coupled with the API documentation using Swagger functionality from Camel, you can easily import your APIs into Apiman, and use its gateways to full control access to your REST services using a breath of control mechanism. This becomes even easier on a container based platform such as Kubernetes or OpenShift where Camel and Apiman can play together seamless. We will talk about this in chapter 18.

For Web Services we have the following to highlight:

- *Use the CXF component for all your web services needs.* The CXF component allows you to make calls to a variety of web services types or to expose your Camel route to the world as a web service. Apache CXF is an active and maintained project with a diverse community and commercial companies supporting the project.

- *Apache CXF and the CXF component is very configurable.* Apache CXF is a very old and mature Web Services project. CXF implements a lot of WS-* specifications and offers a breath of configuration and security options.
- *Only use Web Services if really really needed.* Web Services are no longer a modern technology and you should only use it to integrate with legacy systems. And did we say you should only use it if really needed? We really really really mean that ;)

We have now covered a lot of grounds about services with RESTful and Web Services. We'll now take a leap into another world, one that's often tacked 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.

11

Error Handling

This chapter covers

- Understanding error handling
- Where and when Camel's error handling applies
- The different error handlers in Camel
- Using redelivery policies
- Handling exceptions with `onException`
- Reusing error handlers in all your routes
- Fine-grained control of error handling

We have now reached the half way mark of the book, and we have covered a lot of grounds, but what we are 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 are 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 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 only reacts under certain conditions.

The authors believe this chapter covers a complicated topic that users of Camel may find themselves in trouble learning. Therefore we have taken time and space to cover all aspects of this in details, and as a result this chapter is a long chapter, that is not a light read. Therefore we suggest you take a break half way - we will tell you when.

11.1 Understanding error handling

Before jumping into the world of error handling with Camel, we need to take a step back and look at errors more generally. There are two main categories of errors, recoverable and irrecoverable, and we need to look at where and when error handling starts, because there are some prerequisites that must happen beforehand. Recoverable and irrecoverable errors

When it comes to errors, we can divide them into *recoverable* and *irrecoverable* errors, as illustrated in figure 11.1.

An *irrecoverable error* is an error that 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 an `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 a `java.io.IOException`. On a subsequent attempt, the network issue could be resolved and your application could continue to operate.

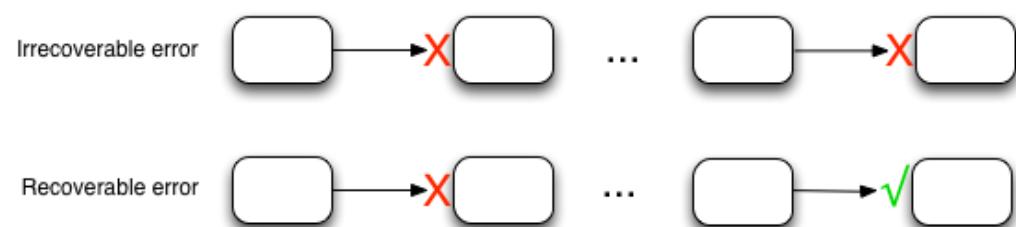


Figure 11.1 Errors can be categorized as either recoverable or irrecoverable. Irrecoverable errors continue to be errors on subsequent attempts; recoverable errors may be quickly resolved on their own.

In your daily life as a Java developer, you won't encounter this division of errors into recoverable and irrecoverable often. 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, where 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) { ①
        try {
            service.sendOrder(order);
            done = true;
        } catch (Exception e) {
            if (--retries == 0) {
                throw new OrderFailedException(e);
            }
        }
    }
}
```

① Attempts redelivery

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` using one of the following three 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 API compatibility. The 2nd `getException` method accepts a class type which allows to traverse the exception hierarchy to find a matching exception. We cover how this works in section 11.4.

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.

Note So why are the two types of errors represented differently? There are 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 non recoverable errors as well, for example using exception classes that by nature are non recoverable. For example a `InvalidOrderIdException` represents an non recoverable error, as a given order id is invalid. It is often more common with Camel to use this approach over fault messages. Fault messages are often only used with legacy components such as JBI / WebServices which by nature has the fault message concept as well.

Now that you've seen recoverable and irrecoverable errors in action, let's summarize how they're represented in Camel:

- Exceptions are represented as recoverable errors.
- Fault messages are represented as irrecoverable errors.

Now let's look at when and where Camel's error handling applies.

11.1.1 Where Camel's error handling applies

Camel's error handling doesn't apply everywhere. To understand why, take a look at figure 11.2.

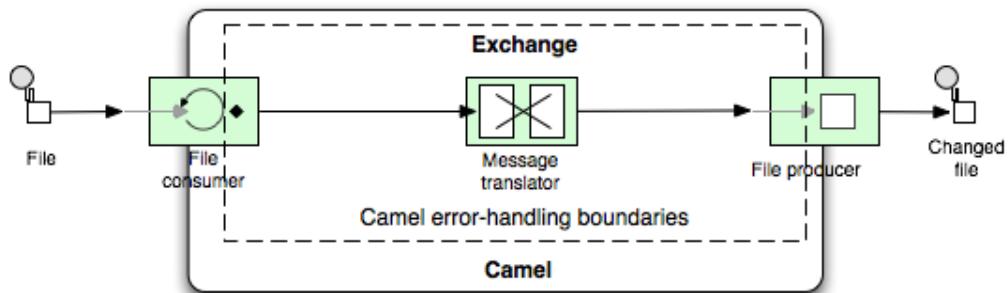


Figure 11.2 Camel's error handling only applies within the lifecycle of an exchange. The error handler boundaries is represented as the dashed square in the figure.

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.

So 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 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 different? 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 we will learn how to deal with when we have more experience with Camel's error handler. We will all come back to these questions and get answers in section 11.4.8 towards the end of this chapter. What you need to understand for now is that figure 11.2 represents the default behavior of 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, we'll start by looking at the different error handlers Camel provides.

11.2 Error handlers in Camel

In the previous section you learned that Camel regards all exceptions as recoverable and stores them on the exchange using the `setException(Throwable cause)` method. This means error handlers in Camel will only react to exceptions set on the exchange. The rule of thumb is that error handlers in Camel only trigger when `exchange.getException() != null`.

Camel provides a range of error handlers. They're listed in table 11.1.

Table 11.1 The 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 following chapter 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 DeadLetterChannel 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 x.y all extend the `RedeliveryErrorHandler` class. That class contains the majority of the error-handling logic that the first three error handlers all leverage. 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 The default error handler

Camel is preconfigured to use the `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. So what happens if an exception is thrown from the `validate` method on the order service bean?

To answer this, we need to dive into Camel's inner processing, where the error handler lives. In every Camel route, there is a Channel that sits between each node in the route graph, as illustrated in figure 11.3.

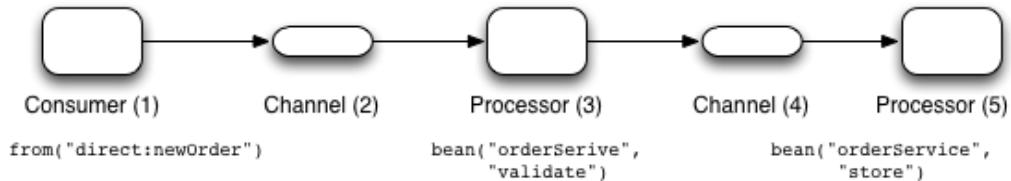


Figure 11.3 A detailed view of a route path, where channels act as controllers between the processors

The Channel is in between each node of the route path, which ensures it can act as a controller that monitors and controls 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.

CamelInternalProcessor

The Channel is implemented in the class `org.apache.camel.processor.CamelInternalProcessor`. The principle within this class is that it performs a series of work before and after processing each step in the routes, which are defined by the `CamelInternalProcessorAdvice` interface. Each functionality such as message tracing, interceptors and the likes is then implemented as an advice. If you ever find yourself in need for deep level Camel debugging, then we have included code comments in the `CamelInternalProcessor` class how to quicker debug this class, when you want to debug through your Camel routes.

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 ③ would throw an exception, which would be propagated back to the previous channel ②, 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
- Exceptions are propagated back to the caller
- The stacktrace 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, then the default behavior is to log a route history that traces the steps back the Exchange was routed. This allows you to quickly identify where in the route the exception occurred. In addition Camel also logs details from the current Exchange such as the message body and headers. We will cover this in more details in section 11.3.3 where we see this in action.

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, which is 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

The 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.

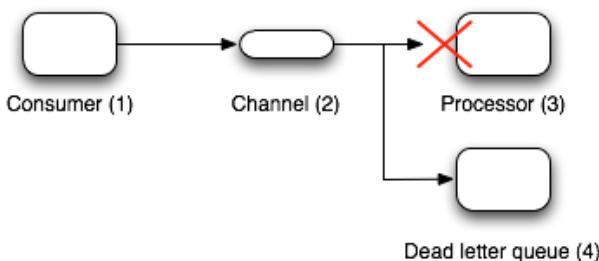


Figure 11.4 The Dead Letter Channel EIP moves failed messages to a dead letter queue.

As you can see, the consumer ① consumes a new message that is 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 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, 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. We will cover this in more details in chapter 12.

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 is how you'd 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 listing:

```
public void configure() throws Exception {
    errorHandler(deadLetterChannel("log:dead?level=ERROR"));           ①

    from("direct:newOrder")
        .bean("orderService", "validate")
        .bean("orderService", "store");
}
```

- ① Configures error handler for all routes in this RouteBuilder class
- ② Configures the routes

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 context or scope level. The following listing is an equivalent of the prior Java DSL listing:

```
<camelContext errorHandlerRef="myErrorHandler">
    <errorHandler id="myErrorHandler" type="DeadLetterChannel" ②
        ①
```

```

        deadLetterUri="log:dead?level=ERROR"/>
<route>
<from uri="direct:newOrder"/>
<bean ref="orderService" method="validate"/>
<bean ref="orderService" method="store"/>
</route>
</camelContext>
```

3

Inside the <camelContext> we configure the error handler ① . As the error handlers in XML DSL can be used in two levels

- Context scoped level
- Route scoped level

A route scope will take precedence over a context scoped error handler. In the code listing we have declare the error handler as a context scope ② by referring to the error handler by its id. The following routes ③ will 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 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 this, 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 would 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 does not include any information about the cause of the error such as the exception message or its stacktrace. The reason is that the caused exception is stored as a property on the Exchange, and exchange properties are not part of the JMS message that Camel sends. To include such information, we would need to enrich the message with such details before its being sent to the `jms:queue:dead` destination.

ENRICHING MESSAGE WITH CAUSE OF ERROR

To enrich the message with the cause of the error we can use a Camel processor, where we implement the logic to enrich the message. And then configure the error handler to use the processor. An example of how to do that is presented in listing 11.1.

Listing 11.1 - Using a processor to enrich the message before its sent to the dead letter channel

```
public class FailureProcessor implements Processor {  
    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);  
    }  
}  
  
errorHandler(deadLetterChannel("jms:queue:dead")  
    .useOriginalMessage().onPrepareFailure(new FailureProcessor()));
```

- ① Processor to enrich the message with details why it failed
- ② Stored the failure reason as a header on the Exchange
- ③ Configuring the error handler to use the processor to prepare the failed Exchange

To enrich the message we can implement this in a `org.apache.camel.Processor` ① . As we use a JMS message queue as the dead letter queue, we must store the failure reason on the Exchange in the message payload - as Exchange properties are not included in JMS messaging. Therefore we use a message header ② to store the failure reason. We then need to configure the DeadLetterChannel error handler to use the processor before the Exchange is send to the dead letter queue ③ .

Configuring the error handler using XML DSL is slightly different, as you need to refer to the Processor using the `onPrepareFailureRef` attribute on the error handler as shown below:

```
<bean id="failureProcessor" class="org.camelinaction.FailureProcessor"/>

<camelContext errorHandlerRef="myErrorHandler" ...>
    <errorHandler id="myErrorHandler" type="DeadLetterChannel"
        deadLetterUri="jms:queue:dead" useOriginalMessage="true"
        onPrepareFailureRef="failureProcessor"/>
    ...
</camelContext>
```

The source code of the book contains this example in the `chapter11/enrich` directory. To try the example use the following Maven goal for either Java or XML version:

```
mvn test -Dtest=EnrichFailureProcessorTest
mvn test -Dtest=SpringEnrichFailureProcessorTest
```

Instead of using a processor to enrich the failed message, you can also use a Camel route, and let the dead letter endpoint call to the route using the direct component. The previous example can be implemented using a route as shown in listing 11.2.

Listing 11.2 - Using a route with a bean to enrich the message before its sent to the dead letter channel

```
public class FailureBean { ①

    public void enrich(@Headers Map headers, Exception cause){
        String failure = "The message failed because " + e.getMessage();
        headers.put("FailureMessage", failure); ②
    }
}

errorHandler(deadLetterChannel("direct:dead").useOriginalMessage()); ③

from("direct:dead")
    .bean(new FailureBean())
    .to("mock:dead"); ④
```

① Bean to enrich the message with details why it failed

② Stored the failure reason as a header on the message

③ Configuring the error handler to use a route to prepare the failed Exchange

④ The route that enriches the Exchange by calling the bean and send the message to the dead letter queue which is mock:dead in this example

We use a Java bean to enrich the message ① . Using a Java bean allows us to leverage Camels parameter binding in the method signature. The first parameter is the message headers which is mapped using the @Headers annotation. The 2nd parameter is the caused exception. We then construct the error message ② which we 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 source code of the book contains this example in the `chapter11/enrich` directory. To try the example use the following Maven goal for either Java or XML version:

```
mvn test -Dtest=EnrichFailureBeanTest
mvn test -Dtest=SpringEnrichFailureBeanTest
```

In this example the route is just calling a bean, but you are free to expand the route to do more work - there is only one consideration. The fact the route is triggered by the error handler, so 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 you have and that is why Camel has a built-in fatal error handler that detects this and break out of this situation. We will cover this in more detail in section 11.4.5.

Let's move on to the transaction error handler.

11.2.3 The transaction error handler

The 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 seldom used and are much simpler.

11.2.4 The no error handler

The 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 the NoErrorHandler.

11.2.5 The logging error handler

The 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 using the log name `org.apache.camel.processor.LoggingErrorHandler` at ERROR level. You can, of course, customize this.

The logging error handler can be replaced by using the dead letter channel error handler and use 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 define policies for whether or not 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 or not 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. |

At this point, you may be eager to see the error handlers in action. In section XXX we'll build a use case that introduces error handling, so there will be plenty of opportunities to try this on your own. But first, let's look at the major features. We'll look at redelivery and scope in section 11.3. Exception policies and error handling will be 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.

In this section, we'll take a look at a real-life error-handling scenario, and then focus on how Camel controls redelivery and how you can configure and use it. We'll also take a look at how you can use error handlers with fault messages. We'll 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 have 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=1h")
    .to("http://riders.com?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 there is no explicit error handling 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 will log the exception and do a file rollback, meaning that any picked-up files will be left on the file system, 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 only move 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 due to 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. The RedeliveryErrorHandler must then know whether or not to attempt redelivery; this is what the redelivery policy is for.

A redelivery policy defines how and whether redelivery should be attempted. Table 11.3 outlines all the options supported by the redelivery policy and what the default settings are.

Table 11.3 Options provided in Camel for configuring redelivery

| Option | Type | default | Description |
|------------------------------|---------|---------|---|
| AllowRedeliveryWhileStopping | boolean | true | Controls whether the error handler is allowed to perform redelivery attempts while the route or CamelContext is being stopped. Setting this option to false, will not attempt any redelivery but immediately exhaust the exchange. |
| AsyncDelayedRedelivery | boolean | false | Dictates whether or not Camel should use asynchronous delayed redelivery. When a redelivery is scheduled to be redelivered in the future, Camel would normally have to block the current thread until it's time for redelivery. By enabling this option, you let Camel use a scheduler so that an asynchronous thread will perform the redelivery. This ensures that no thread is blocked while waiting for redelivery. |
| BackOffMultiplier | double | 2.0 | Exponential backoff multiplier used to multiply each consequent delay. RedeliveryDelay is the starting delay. Exponential backoff is disabled by default. |
| CollisionAvoidanceFactor | double | 0.15 | A percentage to use when calculating a random delay offset (to avoid using the same delay at the next attempt). Will start with the RedeliveryDelay as the starting delay. Collision avoidance is disabled by default. |
| DelayPattern | String | - | A pattern to use for calculating the delay. The pattern allows you to specify fixed delays for interval groups. For example, the pattern "0:1000; 5:5000;10:30000" will use a 1 second delay for redelivery attempts 1 to 5, 5 seconds for attempts 6 to 10, and 30 seconds for subsequent attempts. |
| ExchangeFormatterRef | String | - | Refers to using a custom org.apache.camel.spi.ExchangeFormatter that generates the log message from the Exchange. |
| LogContinued | boolean | false | Specifies whether or not the continuation of redelivery attempts (when all redelivery attempts have failed) should be logged. |

| | | | |
|----------------------------|--------------|-------|--|
| LogExhausted | boolean | true | Specifies whether or not the exhaustion of redelivery attempts (when all redelivery attempts have failed) should be logged. |
| LogExhaustedMessageHistory | boolean | - | Specifies whether or not the message history should be logged when logging exhausted. This option is default true for all error handlers, except the dead letter channel where its false. |
| LogHandled | boolean | false | Specifies whether or not handled exceptions should be logged. |
| LogNewException | boolean | true | Specifies whether new exceptions should be logged during handling of a previous exception. |
| LogRetryAttempted | boolean | true | Specifies whether or not redelivery attempts should be logged. |
| LogRetryStackTrace | boolean | false | Specifies whether or not stacktraces should be logged when a delivery has failed. |
| LogStackTrace | boolean | true | Specifies whether or not stacktraces should be logged when all redelivery attempts have failed. |
| MaximumRedeliveries | int | 0 | Maximum number of redelivery attempts allowed. 0 is used to disable redelivery, and -1 will attempt redelivery forever until it succeeds. |
| MaximumRedeliveryDelay | long | 60000 | An upper bound in milliseconds for redelivery delay. This is used when you specify non-fixed delays, such as exponential backoff, to avoid the delay growing too large. |
| RedeliveryDelay | long | 1000 | Fixed delay in milliseconds between each redelivery attempt. |
| RetriesExhaustedLogLevel | LoggingLevel | ERROR | Log level used when all redelivery attempts have failed. |
| RetryAttemptedLogLevel | LoggingLevel | DEBUG | Log level used when a redelivery attempt is performed. |
| UseExponentialBackoff | boolean | false | Specifies whether or not exponential backoff 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 backoff, and have Camel log at WARN level when it attempts a redelivery, you could use this code:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(5)
        .retryAttemptedLogLevel(LoggingLevel.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 we configure the redelivery policies 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 is 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 the `DefaultErrorHandler`, it's time to try a little example.

11.3.3 Using the `DefaultErrorHandler` with redelivery

In the source code for the book, you'll see an example in the `chapter11/errorhandler` directory. The example uses the following route configuration:

```
errorHandler(defaultErrorHandler()                                     ①
    .maximumRedeliveries(2)
    .redeliveryDelay(1000)
    .retryAttemptedLogLevel(LoggingLevel.WARN));

from("seda:queue.inbox")
    .bean("orderService", "validate")                                ②
    .bean("orderService", "enrich")
    .log("Received order ${body}")
    .to("mock:queue.order");
```

- ① Configures error handler
- ② Invokes enrich method

This configuration first defines a context-scoped error handler ① that will attempt at most two redeliveries using a 1-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 is the example using XML DSL instead:

```
<camelContext errorHandlerRef="myErrorHandler">
    <errorHandler id="myErrorHandler" type="DefaultErrorHandler">
        <redeliveryPolicy 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 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:

```

20015-03-08 14:28:16,959 [a://queue.inbox] WARN DefaultErrorHandler      - Failed
    delivery for exchangeId: 64bc46c0-5cb0-4a78-a4a8-9159f5273601. On delivery attempt: 0
    caught: camelinaction.OrderException: ActiveMQ in Action is out of stock
2015-03-08 14:28:17,960 [a://queue.inbox] WARN DefaultErrorHandler      - Failed
    delivery for exchangeId: 64bc46c0-5cb0-4a78-a4a8-9159f5273601. 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 this at the point of origin. In the preceding example the error occurred when invoking the `enrich` method ②, which means Camel will redeliver by retrying the `.bean("orderService", "enrich")` step in the route.

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:

```

2015-03-08 14:28:18,961 [a://queue.inbox] ERROR DefaultErrorHandler      - Failed
    delivery for exchangeId: 64bc46c0-5cb0-4a78-a4a8-9159f5273601. 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 till this error, as shown below:

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 we can see the exception occurred at the last step which is at id bean4, which would be calling the orderService bean and the enrich method.

Below the message history Camel prints information from the Exchange, as shown:

Exchange

```
Exchange[
  Id ID-davsclaus-air-56685-1426188973293-1-3
  ExchangePattern InOnly
  Headers {breadcrumbId=ID-davsclaus-air-56685-142618893-1-1,
CamelRedelivered=true, CamelRedeliveryCounter=2, CamelRedeliveryMaxCounter=2}
  BodyType String
  Body amount=1,name=ActiveMQ in Action,id=123
]
```

And further below is the full stacktrace of the caused exception, as shown below:

Stacktrace

```
camelinaction.OrderException: ActiveMQ in Action is out of stock
  at camelinaction.OrderService.enrich(OrderService.java:41)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
  at
  sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke(Method.java:606)
  at org.apache.camel.component.bean.MethodInfo.invoke(MethodInfo.java:408)
```

As you can see there is detailed information from Camel when an exception occurs, and the exception cannot be mitigated by the error handler (all redelivery attempts have failed and the Exchange is regarded as exhausted).

NOTICE Unlike the default error handler, the dead letter channel will not log any activity by default. However you can configure the dead letter channel to do so, such as setting `logExhaustedMessageHistory=true` to enable the detailed logging.

The level of information can be turned down, so its only the exception being logged, by setting the option `log exhausted message history` to false as shown below:

```
errorHandler(defaultErrorHandler()
  .maximumRedeliveries(2)
  .redeliveryDelay(1000)
  .retryAttemptedLogLevel(LoggingLevel.WARN)
  .logExhaustedMessageHistory(false);
```

And here is 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.

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 |
|-------------------------------|---------|--|
| Exchange.REDELIVERY_COUNTER | int | The current redelivery attempt. |
| Exchange.REDELIVERED | boolean | Whether this Exchange is being redelivered. |
| Exchange.REDELIVERY_EXHAUSTED | boolean | Whether this Exchange has attempted (exhausted) all redeliveries and has still failed. |

The information in table 11.4 is only available 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.

USING ASYNCHRONOUS DELAYED REDELIVERY

In the previous example, the error handler was configured to use delayed redelivery with a 1-second delay between attempts. When a redelivery is to be conducted, Camel will wait for 1 second before carrying out the redelivery.

If you look at the console output, you can see the redelivery log entries are 1 second apart, and it's the same thread processing the attempts; this can be identified by the [a://queue.inbox] being logged. This is known as synchronous delayed redelivery. There will also be situations where you want to use asynchronous delayed redelivery. So what does that mean?

Suppose two orders are sent to the `seda:queue.inbox` endpoint. The consumer will pick up the first order from the queue and process it. If it fails, it's scheduled for redelivery. In the synchronous case, the consumer thread is blocked while waiting to carry out the redelivery. This means the second order on the queue can only be processed when the first order has been completed.

This isn't the case in asynchronous mode. Instead of the consumer thread being blocked, it will break out and be able to pick up the second order from the queue and continue processing it. This helps achieve higher scalability because threads aren't blocked and doing nothing. Instead the threads are being put to use servicing new requests.

However this comes with a cost. Each asynchronous scheduled redelivery is using an internal in memory queue to hold the scheduled messages, which will have constraints on your system in terms of memory usage.

Camel's error handler using redelivery is not intended to handle long lasting redelivery attempts that span days. It's intended for short term tasks that are spanning a time range in seconds or minutes and up till a hour the most. For longer redelivery attempts its better practice to re-design your application to offload the messages to a persistent store, that Camel can use as source to transport the messages again.

TIP We'll cover the threading model in chapter 13, which will explain how Camel can schedule redeliveries for the future to be processed by other threads. The Delayer and Throttler EIPs have similar asynchronous delayed modes, which you can leverage by enabling the `asyncDelayed` option.

The source code for the book contains an example that illustrates the difference between synchronous and asynchronous delayed redelivery, in the chapter11/errorhandler directory. You can try it using the following Maven goal:

```
mvn test -Dtest=SyncVSAsyncDelayedRedeliveryTest
```

The example contains two methods: one for the synchronous mode and another for the asynchronous.

The console output for the synchronous mode should be displayed in the following order:

```
[a://queue.inbox] INFO - Received input amount=1,name=ActiveMQ in Action
[a://queue.inbox] WARN - Failed delivery for exchangeId: xxxx
[a://queue.inbox] WARN - Failed delivery for exchangeId: xxxx
[a://queue.inbox] WARN - Failed delivery for exchangeId: xxxx
[a://queue.inbox] INFO - Received input amount=1,name=Camel in Action
[a://queue.inbox] INFO - Received order amount=1,name=Camel in Action,id=123,status=OK
```

Compare that with the following output from the asynchronous mode:

```
[a://queue.inbox] INFO - Received input amount=1,name=ActiveMQ in Action
[a://queue.inbox] WARN - Failed delivery for exchangeId: xxxx
[a://queue.inbox] INFO - Received input amount=1,name=Camel in Action
[a://queue.inbox] INFO - Received order amount=1,name=Camel in Action,id=123,status=OK
[rRedeliveryTask] WARN - Failed delivery for exchangeId: xxxx
[rRedeliveryTask] WARN - Failed delivery for exchangeId: xxxx
```

Notice how the *Camel in Action* order is processed immediately when the first order fails and is scheduled for redelivery. Also pay attention to the thread name that executes the redelivery, identified by `[rRedeliveryTask]` being logged. As you can see, it's not the consumer anymore; its a redelivery task.

11.3.4 Error handlers and scopes

Scopes can be used to define error handlers at different levels. Camel supports in two scopes: a context scope and a route scope.

IMPORTANT When using Java DSL then context scope is implemented as a per RouteBuilder scope, but can become a global scope by letting all RouteBuilder inherit from a base class where the context scoped error handler is configured. We 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 for a particular route. This is illustrated in listing 11.3.

Listing 11.3 Using two error handlers at different scopes

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(2)
    .redeliveryDelay(1000)
    .retryAttemptedLogLevel(LogLevel.WARN));1

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
        .maximumRedeliveries(5).retryAttemptedLogLevel(LogLevel.INFO)
        .redeliveryDelay(250).backOffMultiplier(2))
    .beanRef("orderService", "validate")
    .beanRef("orderService", "enrich")3
    .to("mock:queue.order");
```

- ① Defines context-scoped error handler
- ② Defines route-scoped error handler
- ③ Invokes enrich method

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 Dead Letter Channel that will send failed messages to a log. Notice that it has different options configured than the former error handler.

The source code for the book includes this example, which you can run using the following Maven goal from the chapter11/errorhandler directory:

```
mvn test -Dtest=RouteScopeTest
```

This example should fail for some messages when the `enrich` ③ method is invoked. This demonstrates how the route-scoped error handler is used as error handler.

The most interesting part of this test class is the `testOrderActiveMQ` method, 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 backoff, 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.

```
2015-03-08 17:03:44,534 [a://queue.inbox] INFO DeadLetterChannel - Failed
    delivery for exchangeId: e80ed4ba-12b3-472c-9b35-31beed4ff51b. On delivery attempt: 5
    caught: camelinaction.OrderException: ActiveMQ in Action is out of stock
2015-03-08 17:03:44,541 [a://queue.inbox] INFO DLC -
    Exchange[BodyType:String, Body:amount=1,name=Action,id=123]
2015-03-08 17:03:44,542 [a://queue.inbox] ERROR DeadLetterChannel - Failed
    delivery for exchangeId: e80ed4ba-12b3-472c-9b35-31beed4ff51b. 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 we've looked at in this section have used the Java DSL. Let's take a 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 as shown in listing 11.4.

Listing 11.4 Using error handling with XML DSL

```
<bean id="orderService" class="camelinaction.OrderService"/>
<camelContext id="camel" errorHandlerRef="defaultEH" ①
    xmlns="http://camel.apache.org/schema/spring">
    <errorHandler id="defaultEH" ②
        <redeliveryPolicy maximumRedeliveries="2" redeliveryDelay="1000"
            retryAttemptedLogLevel="WARN"/>
    </errorHandler>
    <errorHandler id="dlc"
        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"/>
    </route>
```

```

<to uri="mock:file"/>
<to uri="seda:queue.inbox"/>
</route>
<route errorHandlerRef="dlc">④
  <from uri="seda:queue.inbox"/>
  <bean ref="orderService" method="validate"/>
  <bean ref="orderService" method="enrich"/>
  <to uri="mock:queue.order"/>
</route>
</camelContext>
```

- ① Specifies context-scoped error handler
- ② Sets up context-scoped error handler
- ③ Sets up route-scoped error handler
- ④ Specifies route-scoped error handler

To use a context-scoped error handler in XML DSL, you must configure it 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 is 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 have covered how to use error handlers in Java and XML DSL, but there is a subtle difference 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 a base class as your `RouteBuilder` to reuse context scoped error handlers. The following section highlights the difference.

11.3.5 Reusing context scoped error handlers

Reusing context scoped error handler in XML DSL is straightforward as you would expect. Listing 11.5 shows how easy that is.

Listing 11.5 Reusing context scoped error handler in XML DSL

```

<bean id="orderService" class="camelinaction.OrderService"/>
<camelContext id="camel" errorHandlerRef="defaultEH"①
  xmlns="http://camel.apache.org/schema/spring">
  <errorHandler id="defaultEH"②
    <redeliveryPolicy maximumRedeliveries="2" redeliveryDelay="1000"
      retryAttemptedLogLevel="WARN"/>
  </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>
    <from uri="seda:queue.inbox"/>
    <bean ref="orderService" method="validate"/>
    <bean ref="orderService" method="enrich"/>
    <to uri="mock:queue.order"/>
</route>
</camelContext>

```

- ➊ Specifies context-scoped error handler
- ➋ Sets up context-scoped error handler
- ➌ Route using context-scoped error handler out of the box
- ➍ Another route using context-scoped error handler out of the box

In XML DSL we configure context scoped error handler on the `<camelContext>` using the `errorHandlerRef` attribute ➊. The error handler is then configured using the `<errorHandler>` tag ➋. And all routes ➌ ➍ will automatic use the context scoped error handler by default.

In Java DSL the situation is a bit different, as 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 listing 11.6.

Listing 11.6 Reusing context scoped error handler in Java DSL

```

public abstract class BaseRouteBuilder extends RouteBuilder { ➊

    public void configure() throws Exception {
        errorHandler(deadLetterChannel("mock:dead")
            .maximumRedeliveries(2)
            .redeliveryDelay(1000)
            .retryAttemptedLogLevel(LogLevel.WARN));
    }
}

public class InboxRouteBuilder extends BaseRouteBuilder { ➋

    public void configure() throws Exception {
        super.configure(); ➌

        from("file://target/orders?delay=10000")
            .bean("orderService", "toCsv")
            .to("mock:file")
            .to("seda:queue.inbox");
    }
}

public class OrderRouteBuilder extends BaseRouteBuilder { ➏

    public void configure() throws Exception {
        super.configure(); ➐
    }
}

```

```

        from("seda:queue.inbox")
            .bean("orderService", "validate")
            .bean("orderService", "enrich")
            .to("mock:queue.order");
    }
}

```

- ➊ Base class to hold the reused error handler
- ➋ The context-scoped error handler to reuse
- ➌ RouteBuilder extending the base class
- ➍ Calling super.configure() to reuse the context-scoped error handler
- ➎ Route which will use the context-scoped error handler
- ➏ Another RouteBuilder extending the base class
- ➐ Calling super.configure() to reuse the context-scoped error handler
- ➑ Route which also will use the context-scoped error handler

In Java DSL we would need to use Object Oriented principles such as inheritance to reuse context-scoped error handler. Therefore we create a base class ➊ where we configure the context-scoped error handler ➋ in the `configure` method. Our `RouteBuilder` classes now must extend the base class ➌ ➏ and call the `super.configure` method ➍ ➐ which calls the parent class that has the error handler to reuse. Each of the routes ➎ ➑ will now use the context-scoped error handler which we defined in the base class ➋.

As you can see the subtle difference is that in Java DSL we 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 chapter11/reuse directory which you can try with the following Maven goal:

```
mvn test -Dtest=ReuseErrorHandlerTest
mvn test -Dtest=SpringReuseErrorHandlerTest
```

For example try to experiment and see what happens if you forget to call `super.configure` in for example the `OrderRouteBuilder` class.

This concludes our discussion of scopes and redelivery. We'll now look at how you can use Camel error handlers to handle faults.

11.3.6 Handling faults

In the introduction to section 11.2, we mentioned that by default the Camel error handlers will only react to exceptions. Because a fault isn't represented as an exception but as a message that has the fault flag enabled, faults will not be recognized and handled by Camel error handlers.

There may be times when you want the Camel error handlers handle faults as well. Suppose a Camel route invokes a remote web service that returns a fault message, and you want this fault message to be treated like an exception and moved to a dead letter queue.

We've implemented this scenario as a unit test, simulating the remote web service using a bean:

```
errorHandler(deadLetterChannel("mock:dead"));
from("seda:queue.inbox")
    .bean("orderService", "toSoap")
    .to("mock:queue.order");
```

Now, imagine that the `orderService` bean returns the following SOAP fault:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:Envelope xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ns3="http://www.w3.org/2003/05/soap-envelope">
    <ns2:Body>
        <ns2:Fault>
            <faultcode>ns3:Receiver</faultcode>
            <faultstring>ActiveMQ in Action is out of stock</faultstring>
        </ns2:Fault>
    </ns2:Body>
</ns2:Envelope>
```

Under normal situations, the Camel error handler won't react when the SOAP fault occurs. To make it do so, you have to instruct Camel by enabling fault handling.

To enable fault handling on the `CamelContext` (context scope), you simply do this:

```
getContext().setHandleFault(true);
```

To enable it on a per route basis (route scope), do this:

```
from("seda:queue.inbox").handleFault()
    .bean("orderService", "toSoap")
    .to("mock:queue.order");
```

Once fault handling is enabled, the Camel errors handlers will recognize the SOAP faults and react. Under the hood, the SOAP fault is converted into an `Exception` with the help of an interceptor.

You can enable fault handling in Spring XML as follows:

```
<route handleFault="true">
    <from uri="seda:queue.inbox"/>
    <bean ref="orderService" method="toSoap"/>
    <to uri="mock:queue.order"/>
</route>
```

The source code for the book contains this example in the `chapter11/errorhandler` directory, which you can try using the following Maven goals:

```
mvn test -Dtest=HandleFaultTest
mvn test -Dtest=SpringHandleFaultTest
```

TIP You can enable fault handling to let Camel error handlers react to faults returned from components that has the fault message concept such as web service component.

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.

At this point its a good time to take a break before continuing, for example to go grab a fresh cup of coffee or tea; or maybe its 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'll use the term `onException` interchangeably with "exception policy."

We'll cover exception policies piece by piece, looking at how they catch exceptions, how they works 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 hierarchy 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. We'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 or not. 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 has not found a match.
3. Camel moves up the exception hierarchy and compares again with `OrderFailedException`. This time there is 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 3, so the error handler will attempt at most 3 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.

NOTE 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, each showing a scenario of how `onException` works.

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 the `ConnectionException`. This allows you to define different strategies for different kinds of exceptions. In this example, it is 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 was configured to perform only 2 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 was 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 the `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 chapter 11/onexception directory:

```
mvn test -Dtest=OnExceptionFallbackTest
```

ONEXCEPTION AND GAP DETECTION

Can Camel do better if there isn't a direct hit? Yes, it can, 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 explains 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 the `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 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 go 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
```

So 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 only seen 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, JMSException.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.JMSException</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 there are a couple of things you need to be aware of 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 the `onException`.

Suppose you want Camel to retry invoking the external TCP service, in case there has been an IO-related error, such as a lost network connection. To do this, you can simply add the `onException` and configure the redelivery policy as you like. In the following example, the redelivery tries at most 5 times:

```
onException(IOException.class).maximumRedeliveries(5);
```

You've already learned that `onException(IOException.class)` will catch those IO-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 the `onException`. Because the delay was not overridden in the `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 only need to configure 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 an `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 the `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 an `IOException` were thrown? I am sure you'll say the answer is 3—the value defined on the error handler. In this case, though, the answer is 0. Camel won't attempt to do 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 why 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 only invoked 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

Listing 11.7 shows how you can 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 was 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 only work in the route in which they're defined. `OnException`, on the other hand, works in both context and route scopes, so you can try revising listing 11.7 using `onException`. This is illustrated in listing 11.8.

Listing 11.8 Using onException in context scope

```
onException(JMSEException.class)
    .handled(true) ①
    .process(new GenerateFailureResponse());

from("netty4:tcp://0.0.0.0:4444?textline=true")
    .process(new ValidateOrderId())
    .to("jms:queue:order.status")
    .process(new GenerateResponse());
```

① Handles all JMSEExceptions

A difference between `doCatch` and `onException` is that `doCatch` will handle the exception, whereas `onException` will, by default, not handle it. That's why you use `handled(true)` ① to instruct Camel to handle this exception. As a result, when a `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 how 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 a `JMSException` 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.

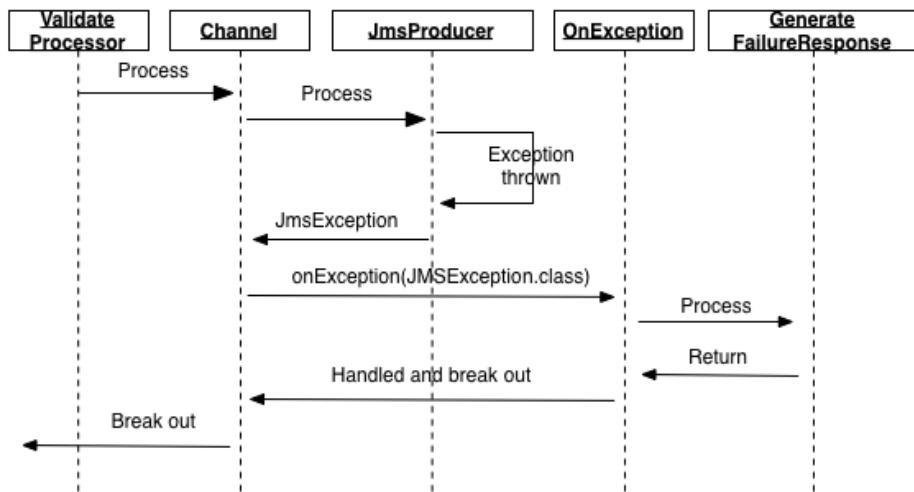


Figure 11.5 Sequence diagram of a message being routed and a `JMSException` being thrown from the `JmsProducer`, which is handled by the `onException`. `OnException` generates a failure that is to be returned to the caller.

Figure 11.5 shows how the `JmsProducer` throws the `JMSException` to the `Channel`, which is where the error handler lives. The route has an `onException` defined that reacts when a `JMSException` is thrown, and it processes the message. The `GenerateFailureResponse` processor generates a custom failure message that is supposed to be returned to the caller. Because the `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 11.4 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 will not continue routing from the point where the exception was thrown. Camel will break out of the route and continue routing on the `onException`. If you want to ignore the exception and continue routing, you must use `continued(true)`, which will be 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.JMSException</exception>
        <handled><constant>true</constant></handled> ①
        <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
      class="camelinaction.FailureResponseProcessor"/>
<bean id="validateOrder" class="camelinaction.ValidateProcessor"/>
<bean id="generateResponse" class="camelinaction.ResponseProcessor"/>

```

① Handles all JMSExceptions

② Processor generates failure response

Notice how `onException` is set up—you must define the exceptions in the `exception` tag. Also, `handled(true)` ① 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 ②. 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 indicates not only what the problem was but that also includes details from the current Camel Message. How can you do that?

Listing 11.9 laid out how to do this using `onException`. Listing 11.10 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); ①
        StringBuilder sb = new StringBuilder();
        sb.append("ERROR: ");
        sb.append(e.getMessage());
        sb.append("\nBODY: ");
        sb.append(body);
        exchange.getIn().setBody(sb.toString());
    }
}

```

➊ Gets the exception

First, you grab the information you need: the message body and the exception ➊ . It may seem a bit odd that you get the exception as a property and not using `exchange.getException()`. You do that because you've marked `onException` to handle the exception; this was done at ➋ in listing 11.9. When you do that, 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, it's only the first two properties in table 11.5 that 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 did not occur while sending to an endpoint, this property is <code>null</code> . |
| <code>Exchange.ERRORHANDLER_HANDLED</code> | <code>Boolean</code> | Whether or not the error handler handled the exception. |
| <code>Exchange.FAILURE_HANDLED</code> | <code>Boolean</code> | Whether or not <code>onException</code> handled the exception. Or <code>true</code> if the <code>Exchange</code> 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 listing 11.11:

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) { ➊
        StringBuilder sb = new StringBuilder();
        sb.append("ERROR: ");
        sb.append(e.getMessage());
        sb.append("\nBODY: ");
        sb.append(body);
        return sb.toString();
    }
}
```

➊ Exception provided as parameter

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

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 was thrown from the processor or bean? This is the topic for the next section.

11.4.5 New exception while handling exception

In this section we will look at how Camel reacts to when a new exception occurs while handling a previous exception, which has been constructed as an example in listing 11.12.

Listing 11.12 Using onException in context scope

```
onException(AuthorizationException.class)           ①
    .handled(true)
    .process(new NotAllowedProcessor());            ②

onException(Exception.class)                      ③
    .handled(true)
    .process(new GeneralErrorProcessor());          ④

public class NotAllowedProcessor implements Processor {

    public void process(Exchange exchange) throws Exception {
        ...
        throw new NullPointerException("null");      ⑤
    }
}

public void GeneralErrorProcessor implement Processor {

    public void process(Exchange exchange) throws Exception {
        ...
        throw new AuthorizationException("forbidden"); ⑥
    }
}
```

- ① onException to handle AuthorizationException
- ② by calling the NotAllowedProcessor
- ③ onException to handle all other kind of Exceptions
- ④ by calling the GeneralErrorProcessor
- ⑤ some code causes a NullPointerException
- ⑥ some code causes a AuthorizationException

We configure two onException ① ③ to handle two different kind of exceptions, by calling a processor ② ④. Each processor implements logic that is executed when either a AuthorizationException or any other kind of exception is thrown. Now the logic in the processors happen to cause a new exception to be thrown ⑤ ⑥. Suppose at first an AuthorizationException is throw during routing, which leads to NotAllowedProcessor to be called, and unfortunately this processor throws a NullPointerException ⑤, which the onException ③ should react and call the GeneralErrorProcessor, which also unfortunately

throws an exception `AuthorizationException`, and so on. This scenario would lead to an endless circular error handling.

To avoid these unfortunate scenarios, Camel does not allow further error handling while already handling an error. In other words, when the `NullPointerException` ⑤ is thrown the first time, then Camel detects that another exception was thrown during error handling, and prevents any further action to take place. This is done by the `org.apache.camel.processor.FatalFallbackErrorHandler` which simply catches the new exception, logs a warning, and sets this as the exception on the `Exchange`, and stops any further routing.

The following code shows a snippet of the warning that is logged by Camel:

```
2015-03-14 12:46:54,885 [main] ERROR FatalFallbackErrorHandler - Exception
    occurred while trying to handle previously thrown exception on exchangeId: ID-
    davsclaus-air-57045-1426333614411-0-2 using:
    [Channel[DelegateSync[camelaction.NotAllowedProcessor@3b938003]]]. The previous and
    the new exception will be logged in the following.
2015-03-14 12:46:54,886 [main] ERROR FatalFallbackErrorHandler - \-->
    Previous exception on exchangeId: ID-davsclaus-air-57045-1426333614411-0-2
    camelaction.AuthorizationException: Forbidden
        at camelaction.NewExceptionTest$1.configure(NewExceptionTest.java:41)
2015-03-14 12:46:54,894 [main] ERROR FatalFallbackErrorHandler - \--> New
    exception on exchangeId: ID-davsclaus-air-57045-1426333614411-0-2
java.lang.NullPointerException
    at camelaction.NotAllowedProcessor.process(NotAllowedProcessor.java:28)
```

You can see this in action by running the following Maven goal from chapter 11/newexception directory:

```
mvn test -Dtest=NewExceptionTest
mvn test -Dtest=SpringNewExceptionTest
```

Instead of handling exceptions there can be situations where you'll want to simply ignore the exception and continue routing. Although those situations comes by rare.

11.4.6 Ignoring exceptions

In section 11.4.3 we learned about how `onException` can handle exceptions. Handling an exception means that Camel will break out of the route. But there can be times when all you want is to catch the exception and continue routing. This is possible to do in Camel using `continued`. All you have to do is to use `continued(true)` instead of `handled(true)`.

Suppose we want to ignore any `ValidationException` which may be thrown in the route, laid out in listing 11.8. Listing 11.13 shows how we can do this.

Listing 11.13 Using continued to ignore ValidationExceptions

```
onException(JMSEException.class)
    .handled(true)
    .process(new GenerateFailureResponse());

onException(ValidationException.class)
```

```
.continued(true); ①

from("netty4:tcp://0.0.0.0:4444?textline=true")
    .process(new ValidateOrderId())
    .to("jms:queue:order.status")
    .process(new GenerateResponse());
```

① Ignores all ValidationExceptions

As you can see, all you have to do is add another `onException` that leverages `continued(true)` ①.

NOTE You can't use both `handled` and `continued` on the same `onException`; `continued` automatically implies `handled`.

Now imagine that a message once again arrives on the TCP endpoint, and the Camel application routes the message. But this time the validate processor throws a `ValidationException`. This situation is illustrated in figure 11.6.

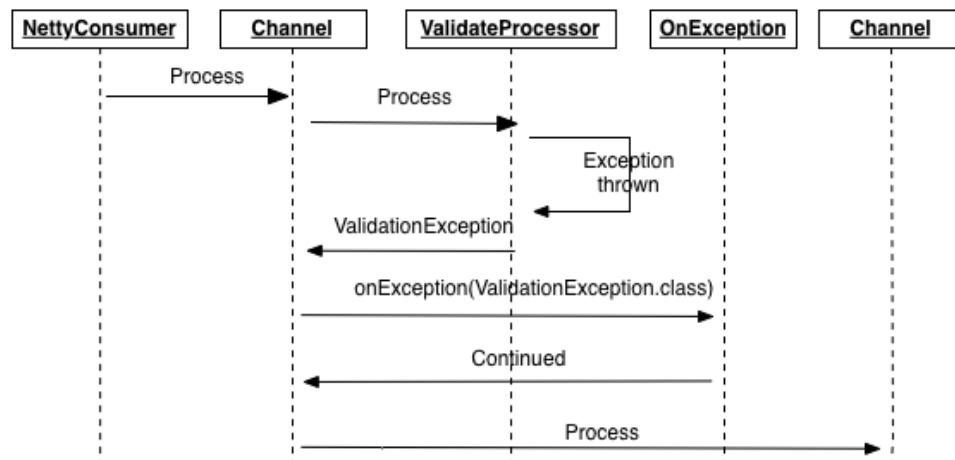


Figure 11.6 Sequence diagram of a message being routed and a `ValidationException` being thrown from the `ValidateProcessor`. The exception is handled and continued by the `onException` policy, causing the message to continue being routed as if the exception were not thrown.

When the `ValidateProcessor` throws the `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 were not 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 have been studying *Camel in Action*, and you've learned that Camel has extensive support for error handling and that you could leverage `onException` to provide this kind of feature. With great confidence, you fire up the editor and alter the route as shown in listing 11.14.

Listing 11.14 Route using error handling with failover to FTP

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(5).redeliveryDelay(10000));

onException(IOException.class).maximumRedeliveries(3)           ①
    .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");
```

① Exception policy

This listing adds an `onException` ① to the route, telling Camel that in the case of an `IOException`, it should try redelivering up to 3 times using a 10-second delay. If there is 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. The `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 ①. The `onException` has been configured to redeliver up till 3 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 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 towards the end of this chapter we have saved a great example for you to try out in practice. The example involves using 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 accompanying source code of the book uses a docker image to use Postgres as the database.

11.4.8 Bridging the consumer with Camel's error handler

In the start of this chapter, in section 11.1.1 we discussed where Camel's error handler works as illustrated in figure 11.2. We learned that Camel's error handler take effect during routing Exchanges. In case of any error happening within the consumer prior to an Exchange being created is to be handled individually by each Camel components. Most of the components will log the error and not create any Exchange to be routed.

To better understand what is being covered in this section, we have include a copy of figure 11.2 in the following as figure 11.7.

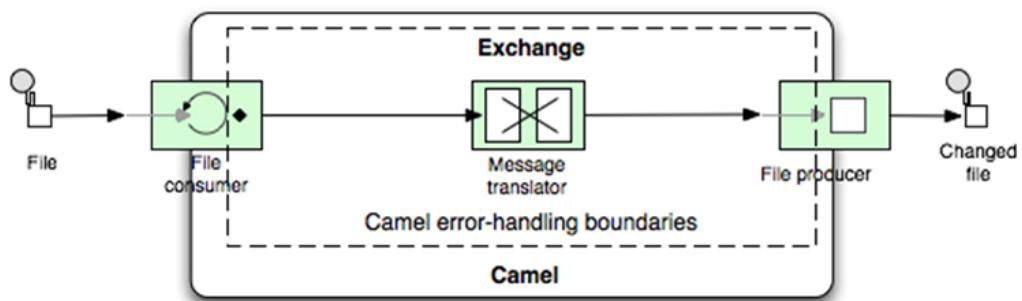


Figure 11.7 Camel's error handling only applies within the lifecycle of an exchange. The error handler boundaries is represented as the dashed square in the figure.

The question unanswered from section 11.1.1 were what happens if the file consumer cannot read the file? We learned that this was component specific and that the file consumer would log a WARN and skip the file, and attempt to read the file again on next poll. So what if we want to handle that error about the file cannot be read, what can we do?

To solve this we 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, then the consumer creates an empty Exchange with the caused error as the exception. The Exchange is then let being 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.

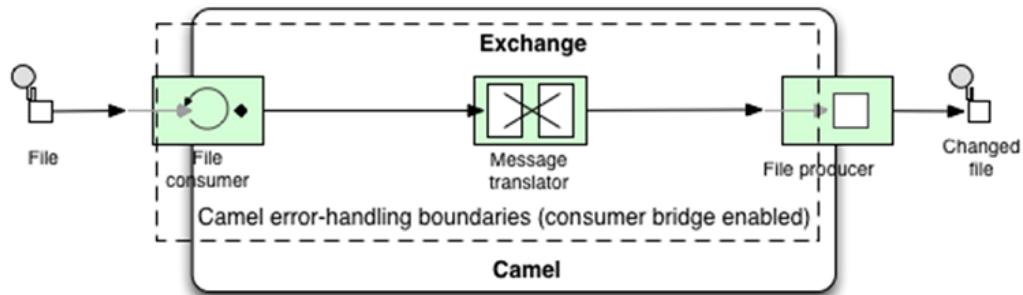


Figure 11.8 Camel's error handling now includes early errors happening from within the consumer, when bridge error handler option has been enabled

Notice from figure 11.8 how Camel's error-handling boundaries is expanded to include the consumer. This means if the consumer captures an exception earlier while attempting to detect or read incoming files, then the exception is captured and send 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 lets use an example to cover this from tail to head. The example is a book order system. A database is used to store incoming orders in a table. Then a Camel application is polling the database table for any rows inserted into the table, and for each row a log message is printed. In case of any error then 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.

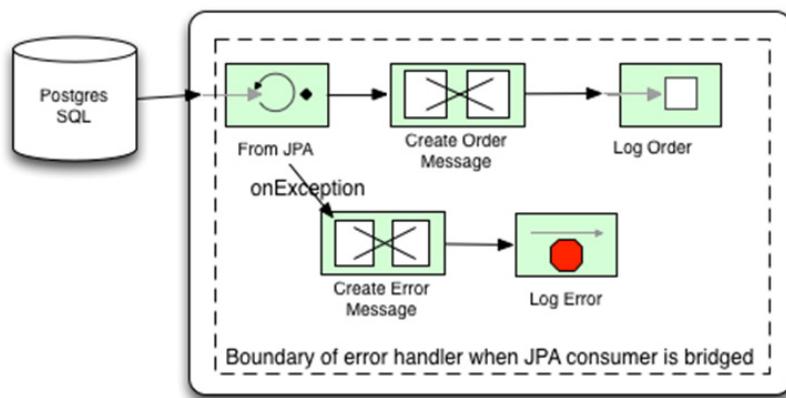


Figure 11.9 - JPA consumer is bridge to Camel's error handler so the onException now also triggers when an error happened internally in the JPA consumer such as no connection to the Postgress SQL database.

The source code of the book contains this example in the chapter11/bridge directory. The example uses docker to run the Postgres database, and there is instructions how to do this from the `readme.md` file from the example source code.

If you have docker setup correctly on your computer, then 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:

```
mvn clean install exec:java
```

The example is constructed so 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 cannot connect to the database, and therefore an error happens internally in the consumer. When you do this then the console should print something similar as shown:

```
2015-05-11 20:08:39,425 [ction.BookOrder] WARN books - Some
exception happened but we do not care Connection to 192.168.59.103:5432 refused. Check
that the hostname and port are correct and that the postmaster is accepting TCP/IP
connections.
```

The error message above tells us that some exception happened but we did not care. And the cause of this error is "Connection to 192.168.59.103:5432 refused". That last part of the error message is a JPA error message when a connection to the database cannot be established - The Postgres database runs on host 192.168.59.103.

If we take a moment to dive into the source code of this example we will learn that the error message above was created by Camel's error handler (④) as shown in listing 11.15

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
        <property key="delay" value="1000"/>
        <property key="consumer.bridgeErrorHandler" value="true"/> ②
        <property key="backoffMultiplier" value="10"/> ③
        <property key="backoffErrorThreshold" value="1"/>
    </endpoint>

    <onException>
        <exception>java.lang.Exception</exception>
        <redeliveryPolicy logExhaustedMessageHistory="false"
                          logExhausted="false"/> ④
        <handled>
            <constant>true</constant>
        </handled>
        <log message="Some exception happened but we do not care ${exception.message}"
             loggingLevel="WARN"/> ⑤
    </onException>
```

```

<route id="books">
    <from uri="ref:newBookOrders"/>
    <log message="Incoming book order ${body.orderId} - ${body.amount} of ${body.title}"/>
</route>
</camelContext>
```

- ➊ Configuring the JPA endpoint outside the route allowing to configure each option using <property> elements
- ➋ Bridging the consumer with Camel's error handler
- ➌ Turn on backoff to multiple the polling delay with x10 after 1 error
- ➍ Tone down logging noise when Camel's error handler is handling the exception
- ➎ Print a logging message about that we do not care about the exception
- ➏ The route that polls the Postgres database

To configure the JPA consumer we setup the endpoint ➊ and provide it with the id "newBookOrders". Each option is configured using a <property> tag. Notice we can of course also configure the endpoint using the uri style that is more common. To bridge the consumer with Camel's error handler, all we have to do is set `consumer.bridgeErrorHandler` to true ➋. In case of an error we do not want the consumer to continue polling as frequent as normal, therefore we apply the backoff multiplier option with a value of 10 ➌. This means that the consumer will instead of polling every 1000 millis, then it will poll 10×1000 millis = 10000 millis (1 sec -> 10 sec). The backoff threshold tells Camel after how many subsequent error's the backoff should apply. In this example after the first error. However if we set the threshold to 5, then only after 5 subsequent errors then the polling would be delayed by the backoff multiplier. When an exception happens, then we want Camel's error handler to react and print a message to the log ➎. And at the bottom of the listing is the route that uses the JPA consumer to pickup new book orders ➏.

TIP We encourage you to try this example on your own. For example try to see what happens when you turn off the `consumer.bridgeErrorHandler`. You could also try changing the backoff thresholds and see what would happen if you remove the `onException` code.

The example used a few options related to backoff, lets just take a moment to present these options formally in a table.

USING BACKOFF TO LET THE CONSUMER BE LESS AGGRESSIVE

Table 11.6 lists all the backoff options that Camel provides out of the box.

Table 11.6 Backoff options to let a Consumer be less aggressive during polling

| Option | Description |
|--------------------------------|--|
| <code>backoffMultiplier</code> | To let the polling consumer backoff if there has 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 is happening again. When this option is in use then |

| | |
|------------------------------------|---|
| | <code>backoffIdleThreshold</code> and/or <code>backoffErrorThreshold</code> must also be configured. |
| <code>backoffIdleThreshold</code> | The number of subsequent idle polls that should happen before the <code>backoffMultiplier</code> should kick-in. |
| <code>backoffErrorThreshold</code> | The number of subsequent error polls (failed due some error) that should happen before the <code>backoffMultiplier</code> should kick-in. |

Before we finish up this chapter, we must 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 Other error-handling features

We'll end this chapter by looking at some of the other features Camel provides for error handling:

- `onWhen`—Allows you to dictate when an exception policy is in use
- `onRedeliver`—Allows you to execute some code before the message is redelivered
- `retryWhile`—Allows you, at runtime, to determine whether or not 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 an `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 listing 11.16.

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
        if (code != 500) {
            return false;
        }
        return "ILLEGAL DATA".equals(cause.getResponseBody().toString());
    }
}
```

① Gets HTTP status code

When the HTTP operation isn't successful, the Camel HTTP component will throw an `org.apache.camel.component.http.HttpOperationFailedException` exception, containing information why it failed. The `getStatusCode()` method on `HttpOperationFailedException` ①, returns the HTTP status code. This allows you to determine if it's an HTTP error code 500 with the "ILLEGAL DATA" body text.

Next, you need to use the utility class from listing 11.16 in your existing route from listing 11.14. But first you add the `onException` to handle the `HttpOperationFailedException` and detour the message to the illegal folder:

```
onException(HttpOperationFailedException.class)
    .handled(true)
    .to("file:/rider/files/illegal");
```

Now, whenever an `HttpOperationFailedException` is thrown, Camel moves the message to the illegal folder.

It would be better if you had more fine-grained control over when this `onException` triggers. How could you incorporate your code from listing 11.16 with the `onException`?

I am sure you have guessed where we're going—yes, you can use the `onWhen` predicate. All you need to do is insert the `onWhen` into the `onException`, as shown here:

```
onException(HttpOperationFailedException.class)
    .onWhen(bean(MyHttpUtil.class, "isIllegalData"))
    .handled(true)
    .to("file:/rider/files/illegal");
```

Camel adapts to your POJO classes and uses them as is, thanks to the power of Camel's parameter binding. This is a powerful way to develop your application without being tied to the Camel API.

Here is how you would use `onWhen` in XML DSL:

```
<onException>
    <exception>org.apache.camel.component.http
        HttpOperationFailedException</exception>
```

The `<exception>` is one line

```
<onWhen><method ref="myHttpUtil" method="isIllegalData"/></onWhen>
<handled><constant>true</constant></handled>
<to uri="file:/rider/files/illegal"/>
</onException>

<bean id="myHttpUtil" class="org.camelinaction.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.

Next, let's look at `onRedeliver`, which allows fine-grained control when a redelivery is about to occur.

11.5.2 Using onRedeliver

The purpose of `onRedeliver` is to allow some 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 an `org.apache.camel.Processor`, in which you implement the code to be executed.

`OnRedeliver` can be configured on the error handler, on `onException`, or on both, as follows:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .onRedeliver(new MyOnRedeliveryProcessor()));

onException(IOException.class)
    .maximumRedeliveries(5)
    .onRedeliver(new MyOtherOnRedeliveryProcessor());
```

`OnRedeliver` is also scoped, so if an `onRedeliver` is set on an `onException`, it overrules any `onRedeliver` set on the error handler.

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.MyOtherOnRedeliveryProceosstor"/>
```

Finally, let's look at one last feature: `RetryWhile`.

11.5.3 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 ruleset that determines how long it should retry. Listing 11.17 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(
        @Header(Exchange.REDELIVERY_COUNTER) Integer counter,
        Exception causedBy) {
        ...
        // return true or false
    }
}
```

Using your own `MyRetryRuleset` class, you can implement your own logic determining whether it should continue retrying or not. If the method returns `true`, a redelivery attempt is conducted; if it returns `false`, it give up.

To use your ruleset, you configure `retryWhile` on the `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 how 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 take away and 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.
- *Use asynchronous delayed redelivery.* If the order of messages processed from consumers doesn't matter, leverage asynchronous redelivery to achieve higher

scalability.

- *Capture error details.* When errors happen try to capture details by ensuring sufficient information is logged.
- *Use monitoring tooling.* Use tooling to monitor your Camel applications so it 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 if your error-handling strategies are up to the task.
- *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.

In this chapter we covered among others the Dead Letter Channel EIP which solution to deal with errors is to move the failure messages to a dead letter queue for later inspection. In the next chapter we talk about how transactions can be used to encapsulates an unit of work as an single atomic work that in case of failure will perform a rollback action, with the end goal, of as if the unit of work never happened.

12

Transactions and Idempotency

This chapter covers

- Why you need transactions
- How to use and configure transactions
- The differences between local and global transactions
- Using transactions with messaging and database
- How to rollback transactions
- How to compensate when transactions aren't supported
- Preventing duplicate messages using idempotency
- Learning about the various idempotent repository implementations that are shipped out of the box

To help explain what transactions are, 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 actually 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 either of two states: either the purchase was accepted and confirmed, or the purchase was 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 I expect you know about the database ACID properties, so I 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 for when you're using resources that don't support transactions. The last part of the chapter covers a topic called idempotency which deals with how to handle duplicate messages. This can often happen when transactions cannot be used, and data exchange between distributed systems may have to replay messages to deal with unreliable network between the systems. The last section of the chapter covers a common use-case how to cluster an application and use clustered idempotency to coordinate work between the nodes, so each node do not process duplicate messages.

12.1 Why use transactions?

There are many good reasons why using transactions makes sense. But before we focus on using transactions with Camel, let's look at what can go wrong when you don't use transactions.

In this section, we'll review an application Rider Auto Parts uses to collect metrics that will be published to an incident management system. We'll see what goes wrong when the application doesn't use transactions, and then we'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 or not their 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.

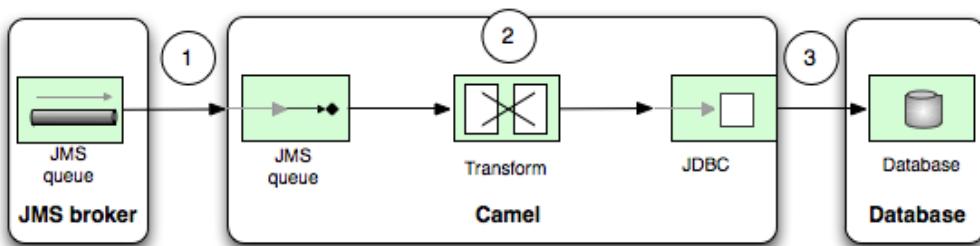


Figure 12.1 Partner reports are received from the JMS broker, transformed in Camel to SQL format, and then written to the database.

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>201611151637</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 following mapping code 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,
                     @XPath("partner/date/text()") String date,
                     @XPath("partner/code/text()") int statusCode,
                     @XPath("partner/time/text()") long responseTime) {
        Map map = new HashMap(); ①
        map.put("id", id);
        map.put("date", date);
        map.put("code", statusCode);
        map.put("time", responseTime);
        return map;
    }
}
```

① Extracts data from XML payload

② Constructs Map with values to insert using SQL

Coding the mapping logic that extracts the data from XML to a `Map` in 10 or so lines in listing 12.1 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 a `Map` to store the input values ②. The SQL `INSERT` statement is externalized from Java code and defined in a properties file which you named `sql.properties`:

```
sql-insert=insert into partner_metric (partner_id, time_occurred, status_code, perf_time)
  values (:#id, :#date, :#code, :#time)
```

Notice how the keys from the Map ❷ matches the values in the SQL statement - such as `:#id` matching `id`, and `:#date` matching `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 the Camel in Action book, we used the Camel JDBC component for this example. However this time we use the SQL component, as the SQL component has been improved since. For example the SQL component uses prepared statements where as JDBC uses regular statements; which should yield in better performance on the database. The authors of the book also admire the `camel-elsql` component that allows to use a light-weight 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)); ❶
    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);
    Thread.sleep(5000);
    assertEquals(1, jdbc.queryForInt(sql)); ❷
}
```

- ❶ Asserts there are no rows in database
- ❷ Asserts one row was inserted into database

This test method outlines the principle. First you check that the database is empty ❶. 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. At the end, you check that the database contains one row ❷.

12.1.2 Setting up the JMS broker and the database

To run this unit test, you need to use a local JMS broker and a database. You can use Apache ActiveMQ as the JMS broker and Derby as the database. Apache Derby can be used as an in-memory database without the need to run it separately. Apache 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. This is shown in listing 12.2.

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"
    class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>

<broker:broker useJmx="false" persistent="false" brokerName="localhost">
    <broker:transportConnectors>
        <broker:transportConnector uri="tcp://localhost:61616"/>
    </broker:transportConnectors>
</broker:broker>

    <bean id="myDataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource"> ③
        <property name="driverClassName"
            value="org.apache.derby.jdbc.EmbeddedXADataSource"/>
        <property name="url" value="jdbc:derby:memory:order;create=true"/>
    </bean>
</beans>
```

- ① Configures ActiveMQ component
- ② Sets up embedded JMS broker
- ③ Sets up database

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`.

Actually, 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 why 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 very good use-case for unit and integration tests, and for some application servers that provides messaging out of the box such as Apache ServiceMix and JBoss Fuse. For other use cases its recommended to run ActiveMQ standalone, which allows to separate the brokers from your applications. ServiceMix and JBoss Fuse users may consider an architecture where they configure some nodes to be dedicated ActiveMQ brokers, and others nodes to host their applications. This allows to cleanly separate brokers from applications; which is a recommended practice. However the one-size-fits-it-all rule apply here, there can be use-cases where co-locating the ActiveMQ brokers with your applications can be good practice such as if the messaging load is light, and you do not 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 listing 12.3.

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
        public void configure() throws Exception {
            interceptSendToEndpoint("sql:*")
                .throwException(new ConnectException("Cannot connect"));
        }
    };
    route.adviceWith(context, rb);                                         ②

    String sql = "select count(*) from partner_metric";
    assertEquals(0, jdbc.queryForInt(sql));                                  ③

    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

- ① Simulates no connection to database
- ② Advises simulation into existing route
- ③ SQL to select number of row in the database
- 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 the `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 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 was not 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*, which means 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 is not very often used with Camel as you (the client) needs to call the `acknowledge` method on the JMS message, to mark the message as successfully consumed; which requires you to write Java code to perform this action. The source code for the book contains an example of using JMS client acknowledge mode in the `chapter12/riderautoparts-partner` directory which you can try 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.

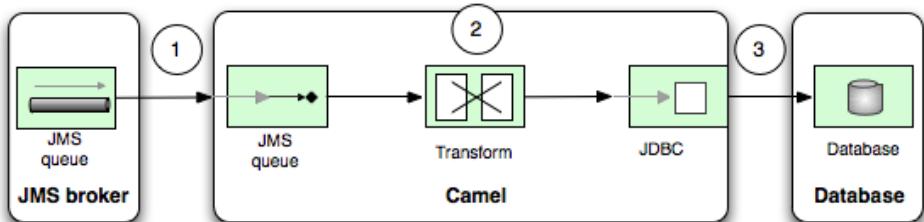


Figure 12.2 A transaction is a series of events between begin and commit.

To demonstrate the sequence in figure 12.2, you could write what are known as locally managed transactions, where 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 was thrown or not.

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 does not reinvent the wheel and implement a transaction manager; which is a complicated piece of technology to build. Instead Camel leverage 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 About 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.

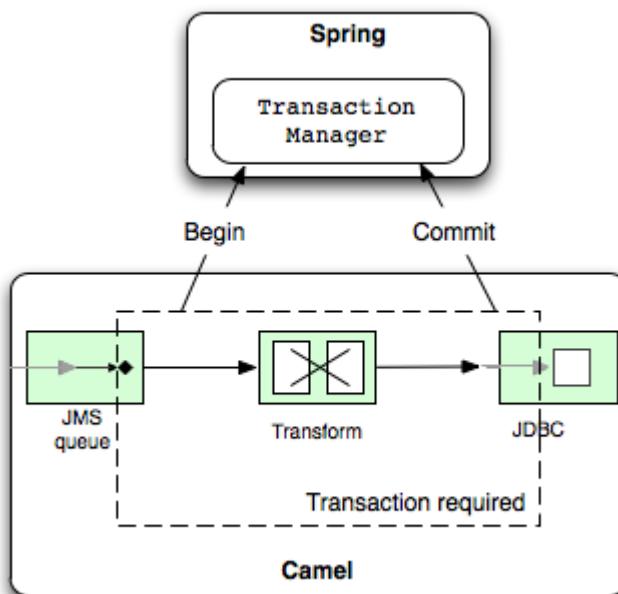


Figure 12.3 Spring's TransactionManager orchestrates the transaction by issuing begins and commits. The entire Camel route is transacted, and the transaction is handled by Spring.

Spring TransactionManager API vs Implementation's

Camel uses the Spring Transaction to manage and orchestrate the transaction using its TransactionManager API. Depending on what kind of resources are taking part in the transaction, an appropriate implementation of the transaction manager must be chosen. Spring offers a number out of the box that works for various local transactions such as JMS and JDBC. However for global transaction you must use a third party JTA transaction manager implementation; JTA transaction manager is provided by JEE application servers. Spring does not offer that out of the box, only the necessary API abstract that Camel uses. We will cover this in the following and the difference between local and global transaction is the topic of section 12.3.1 and 12.3.2.

Figure 12.4 adds more details, to illustrate that the JMS broker also plays an active part in the transaction.

In this figure you can see how the JMS broker, Camel, and the Spring JmsTransactionManager work together. The JmsTransactionManager orchestrates the resources that participate in the transaction ①, which in this example is the JMS broker.

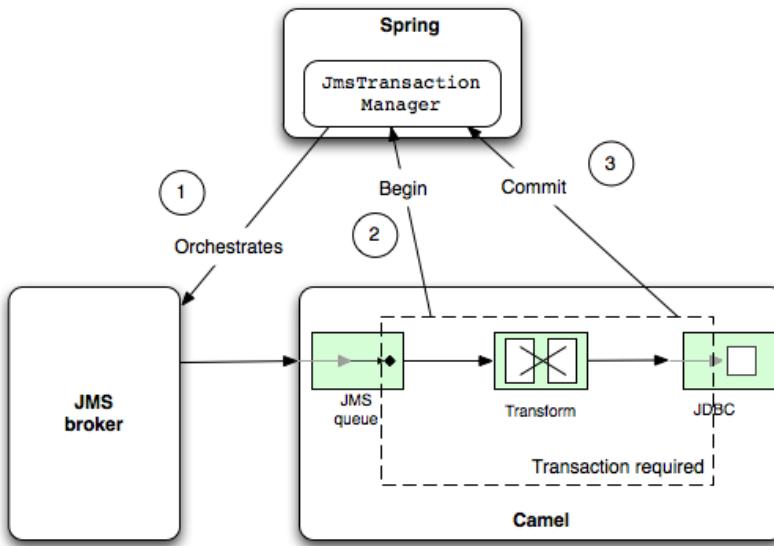


Figure 12.4 The Spring `JmsTransactionManager` orchestrates the transaction with the JMS broker. The Camel route completes successfully and signals the commit to the `JmsTransactionManager`.

When a message has been consumed from the queue and fed into the Camel application, Camel issues a begin ② to the `JmsTransactionManager`. Depending on whether the Camel route completes with success or failure ③ , the `JmsTransactionManager` will ensure the JMS broker commits or rolls back.

It's now time to see how this works in practice. In the next section, we'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 did not 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. Listing 12.4 shows how this is done.

Listing 12.4 XML configuration using Spring transactions

```
<bean id="activemq"
      class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="transacted" value="true"/> ①
    <property name="transactionManager" ref="txManager"/>
    <property name="cacheLevelName" value="CACHE_CONSUMER"/> ②
</bean>
```

```

<bean id="txManager"
      class="org.springframework.jms.connection.JmsTransactionManager">
    <property name="connectionFactory" ref="jmsConnectionFactory"/>
</bean>

<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>

```

- ① Enables transacted acknowledge mode
- ② Enable consumer cache level
- ③ Configures Spring JmsTransactionManager
- ④ URL to the ActiveMQ message broker

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, which is a Spring `JmsTransactionManager` ③ that manages transactions when using JMS messaging.

JMS, JDBC and JTA TransactionManagers

The `JmsTransactionManager` is provided out of the box from Spring JMS component and is only able to handle transaction with JMS resources. In section 12.3.3 we will cover using JDBC as transaction instead of using JMS. And section 12.3.2 covers when we use 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 we enabled `transaction` ① then the default behavior of the JMS component is to let the JMS consumer be in auto caching mode. The caching mode has high impact on the performance, and its highly recommended to configure the mode to be `cache consumer` ② . This can safely be done with most JMS message brokers such as ActiveMQ. Notice when using JTA transactions then the consumer cannot always be cached. This depends on the involved message brokers, databases, and transaction managers, what level of cache can be in use. 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 has a high impact on the performance. Its recommended you read what we just covered one more time.

So far you've only reconfigured beans in the XML file, which is mandatory when using Spring. In Camel, itself, you have not yet configured anything in relation to transactions. Camel offers great convention over configuration for transaction support, so all you have to do is to 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 we will use transactions with only one route, in section 12.3.6 we will go further down the road and cover transaction with multiple routes.

When a route is specified as `<transacted/>` then under the hood, Camel looks up the Spring transaction manager and leverages 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 only applies 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. We'll cover that in section 12.3.5.

Look at the source code to better understand how all this fits together

The source code for this book 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, which may help you better understand how the configuration in the Spring XML file fits together. We will in the following continue and 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. In section 12.3, we'll go 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 source code for this book 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. There is no upfront work needed to install them. To demonstrate how this works, we'll return to the Rider Auto Parts example.

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:

```
2016-10-22 10:08:38,168 [sumer[partners]] WARN TransactionErrorHandler      - Transaction
rollback (0x40fcf6c1) redelivered(false) for (MessageId: ID:davsclaus.air-61646-
1427015316080-7:1:2:1:1 on ExchangeId: ID-davsclaus-air-61645-1427015315762-0-3)
caught: java.net.ConnectException: Cannot connect to the database
2016-10-22 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]
org.apache.camel.RuntimeCamelException: java.net.ConnectException: Cannot connect to the
database
at
org.apache.camel.util.ObjectHelper.wrapRuntimeCamelException(ObjectHelper.java:1619)
at
org.apache.camel.spring.spi.TransactionErrorHandler$1.doInTransactionWithoutResult(Tra
nsactionErrorHandler.java:188)
at
...
at
org.apache.camel.component.jms.EndpointMessageListener.onMessage(EndpointMessageList
ener.java:103)
```

You can tell from the stacktrace that `TransactionErrorHandler` (shown in bold) logged an exception at `WARN` level, with a transaction rollback message. Just below the `EndpointMessageListener` (also shown in bold) 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 will roll back the transaction if an exception is thrown.

So 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. Add the following code at the end of the unit test method with the name `testNoConnectionToDatabase` 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
```

```
camelaction.RiderAutoPartsPartnerTransactedTest.testNoConnectionToDatabase
(RiderAutoPartsPartnerTTest.java:101)
```

We'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, Dejan Bosanac, and Rob Davies.

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 We will cover much more in depth about transaction redeliveries and nuances of such when using ActiveMQ as the message broker in section 12.3.4.

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 only fails 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()
                    .when(header("JMSRedelivered").isEqualTo("false"))
                        .throwException(new ConnectException(
                            "Cannot connect to the database"));
        }
    };
    context.getRouteDefinition("partnerToDB")
        .adviceWith(context, rb);          ②
    NotifyBuilder notify = new NotifyBuilder(context)
        .whenDone(1 + 1).create();         ③
}
```

```

String sql = "select count(*) from partner_metric";
assertEquals(0, jdbc.queryForInt(sql));

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)); ④

assertEquals(1, jdbc.queryForInt(sql));

Object dlq = consumer.receiveBody("activemq:queue:ActiveMQ.DLQ", 1000);
assertNull("Should not be in the DLQ", dlq); ⑤
}

```

- ① Causes failure first time
- ② Advises the route with the simulated error route from ①
- ③ Setup notify builder for how many messages we expect
- ④ Wait for routing to be done
- ⑤ Asserts message not in DLQ

The idea is to throw a `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 or not.

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 ① to test the `JMSRedelivered` header and only throw the exception if it's `false`, which means it's the first delivery. The route must then be advised ② 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 the help from the `NotifyBuilder` we wait for the route to be done ④ expecting two messages in total ③; the first message which should fail, and then the 2nd 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 ⑤.

Example for Apache Karaf users

The RiderAutoParts Partner example that was used in this section is also provided as an example for Apache Karaf / ServiceMix / JBoss Fuse users. The source code for the book 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 example we've just covered uses what are called local transactions, because they're based on using only a single resource in the transaction—Spring was only orchestrating the JMS broker. But there was also the database resource, which, in the example, was not under

transactional control. Leveraging 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.

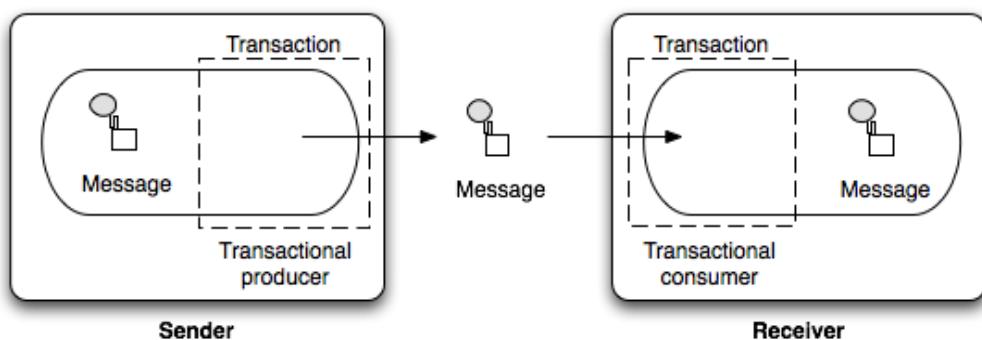


Figure 12.5 A transactional client handles the client's session with the receivers so the client can specify transaction boundaries that encompass the receiver.

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.

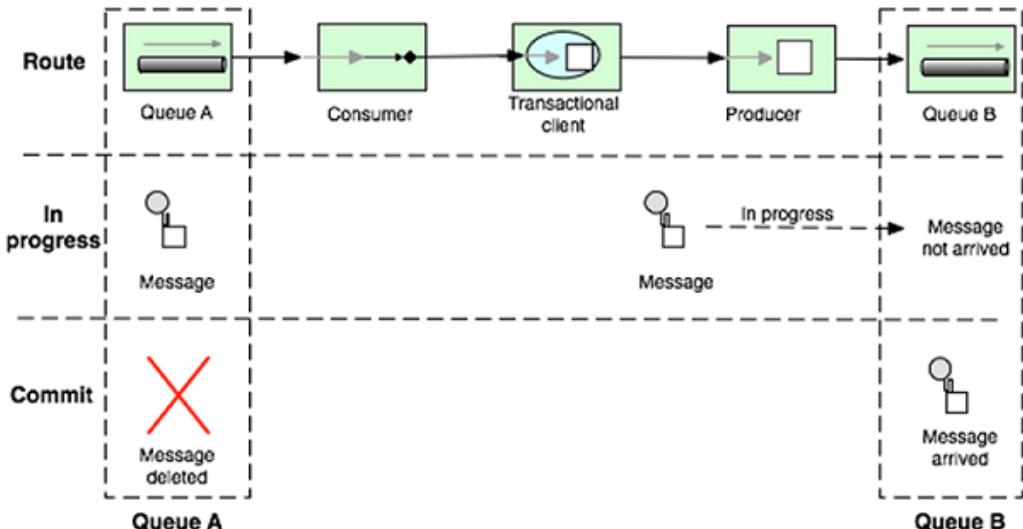


Figure 12.6 A message is being moved from queue A to queue B. Transactions ensure the message is moved in what appears to be an atomic operation.

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 has not yet arrived in queue B. The message stays on 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, we 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, which is the JMS broker.

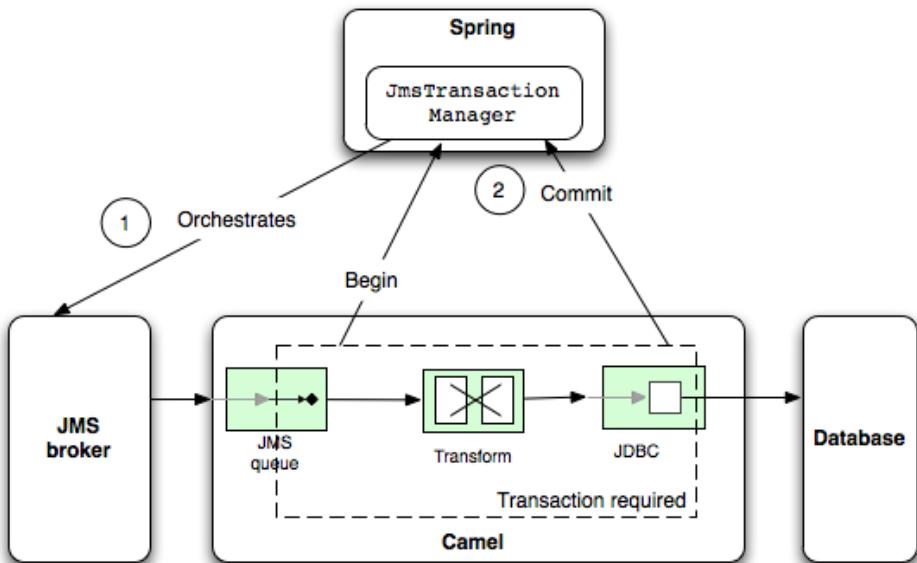


Figure 12.7 Using `JmsTransactionManager` as a single resource in a transaction. The database isn't a participant in the transaction.

In this situation, the `JmsTransactionManager` orchestrates the transaction with the single participating resource, which is the JMS broker (1). The `JmsTransactionManager` from Spring can only orchestrate 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 was because if the database decides to roll back the transaction, it will throw an exception that the Camel `TransactionErrorHandler` propagates back to the `JmsTransactionManager`, which reacts accordingly and issues a rollback (2).

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 the 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.

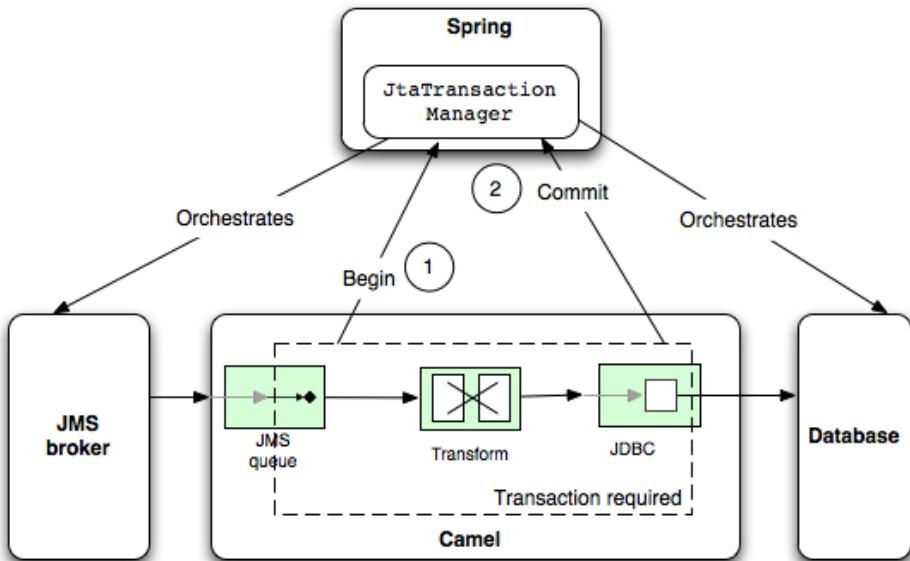


Figure 12.8 Using JtaTransactionManager with multiple resources in a transaction. Both the JMS broker and the database participate in the transaction.

In this figure, we've switched to using the `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 .

So what is the `JtaTransactionManager`, and how is it different from the `JmsTransactionManager` used in the previous section (see 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 leverage 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.

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 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:
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 the ActiveMQ-XAConnectionFactory 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. We can use Atomikos to setup 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, DB2 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 an additional step in the route shown in figure 12.8. You'll process the message *after* it has 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.

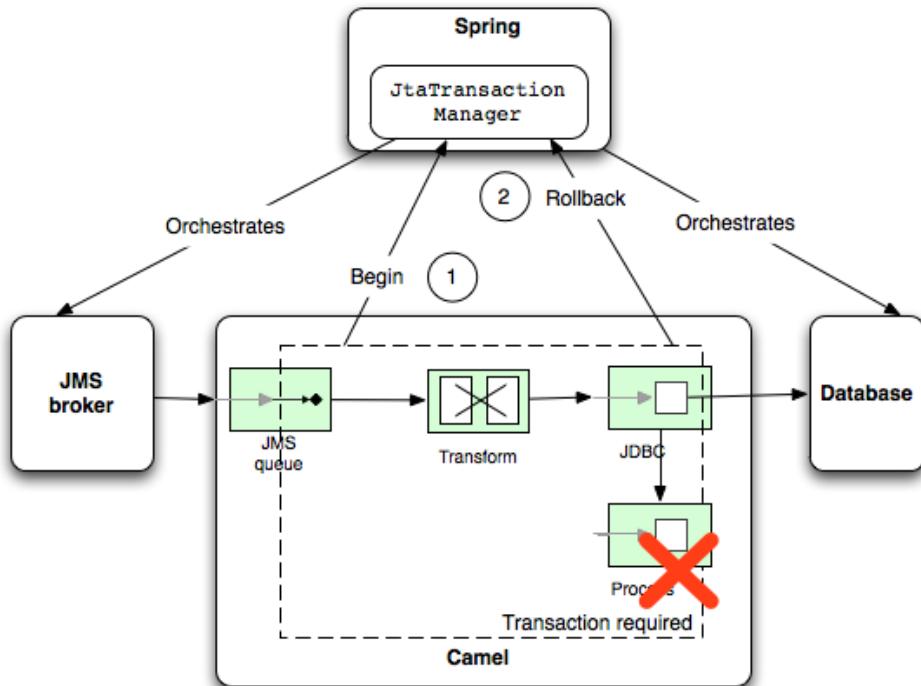


Figure 12.9 A failure to process a message at the last step in the route causes the JtaTransactionManager to issue rollbacks to both the JMS broker and the database.

In figure 12.9, 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. The 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.

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=XA.rollbackBeforeDbTest
mvn test -Dtest=XA.rollbackAfterDbTest
mvn test -Dtest=SpringXACommitTest
mvn test -Dtest=SpringXA.rollbackBeforeDbTest
mvn test -Dtest=SpringXA.rollbackAfterDbTest
```

Apache Karaf example with global transactions

The example is also available for Apache Karaf / ServiceMix / 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 transaction with Apache Karaf is difficult then its highly recommended to study this source code and pay attention to how transaction managers, JCA resources and the likes are defined and configured.

So far all the examples we have covered are starting from a JMS resource. What if its a database that is the starting resource.

12.3.3 Transaction starting from a database resource

Transaction can also begin from a database which is the topic for this section.

Now suppose we flipped the JMS broker and database from the use-case in figure 12.8, which gives us what is illustrated in figure 12.10.

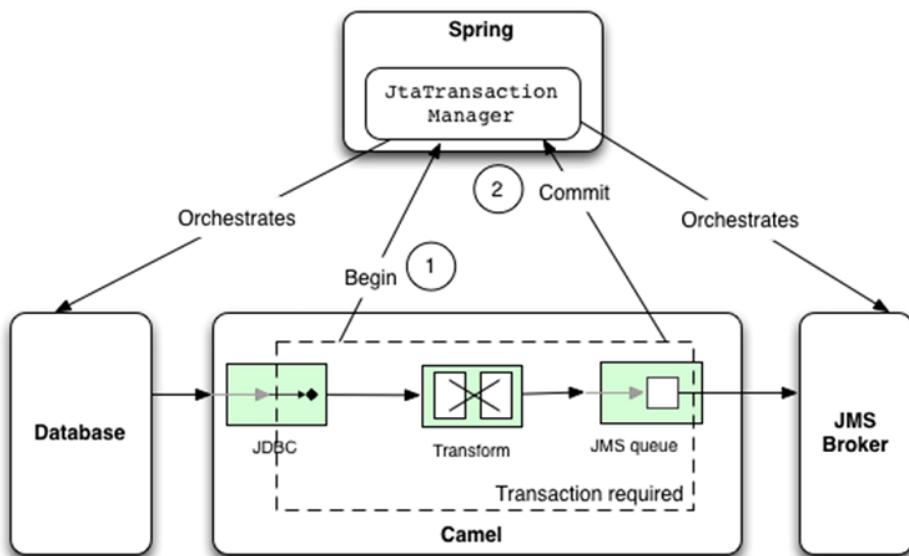


Figure 12.10 Using JtaTransactionManager with multiple resources in a transaction. The transaction starts from a database resource and includes the JMS broker.

Because we are using both a database and a JMS resource in the same transaction we need to use the `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 is changed is the Camel route to start from a database.

The source code of the book contains this example in the chapter12/xa-database directory, which you can try 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 as shown below:

```
<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 transaction being used as well. This is similar to when using JMS, where we configured transacted acknowledge mode as show in listing 12.4.

ONLY A DATABASE

If we take out the JMS Broker from the example so we are only using the database as single resource, then we do not need to use XA, and instead of using the `JtaTransactionManager`, we can use the `DataSourceTransactionManager` which is intended for single resource database. The source code of the book contains such an example in the chapter12/tx-database directory which you can try using the following goals:

```
mvn test -Dtest=SpringCommitTest
mvn test -Dtest=SpringRollbackTest
```

NOTE WildFly users using the resources shipped with the WildFly server is recommended to look at the `wildfly-camel` documentation how to use transactions with WildFly and Apache Camel: http://wildfly-extras.github.io/wildfly-camel/#_jms

When a message is rolled back, then the ACID principle of the transaction means that the outcome was as if the message did not happen - its all or nothing. However what happens next, that is what we will discuss in the next section.

12.3.4 Transaction redeliveries

When a transaction rollback then what happen next depends on whether the message came from a message broker or a database. The former often have various redelivery settings you can configure, which we will cover in this section. However a database, nor the JDBC components from Camel provides transactional redelivery support. This is something we will talk about at the end of this section, what you can do instead.

On a message broker such as Apache ActiveMQ message redelivery can be handled and configured at two different levels:

- Consumer level (is used by default)
- Broker level

By default redelivery is controlled by the consumer, which means its the duty of the consumer to keep track how many times a message has been redelivered, and for how long 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 its in fact also similar to what Camel's error handler is capable of doing, and as such many of the options you use to configure redelivery with ActiveMQ are similar options you would use with Camel. This is not a surprise as both ActiveMQ and Camel was created originally by the same group of people.

To allow the consumer to be able to keep track of redelivery of the message, the consumer must be reused by the Camel route, and to ensure this you must configure the JMS or ActiveMQ component in Camel to cache the consumer as shown in bold below:

```
<bean id="activemq"
    class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="transacted" value="true"/>
    <property name="transactionManager" ref="jtaTransactionManager"/>
    <property name="cacheLevelName" value="CACHE_CONSUMER"/>
</bean>
```

This is important to remember as the JMS or ActiveMQ component will by default be conservative in transacted mode, and not cache the consumer. You may not notice any difference as when a transaction rollback happens, the message is still being redelivered. However as the consumer was not cached, then a new consumer is created for each message received, and that consumer has no previous state about the redeliver, and as a result there is no delays between any redeliveries. Because there is no delay between redeliveries, then the redelivered messages will happen very rapid in sequence. The ActiveMQ also keep tracks of number of redelivery attempts and when the message has failed too many times in a row, its automatic moved to the ActiveMQ Dead Letter Queue.

CONFIGURING CONSUMER LEVEL REDELIVERY WITH ACTIVEMQ

If however the consumer is cached then by default the consumer will delay each redelivery by one second and therefore redeliveries are a bit slower. Because one second may be still to fast, you can configure a slower setting, which can be done in the `brokerURL` option as shown below:

```
<bean id="jmsXAConnectionFactory"
    class="org.apache.activemq.ActiveMQXAConnectionFactory">
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

```
<property name="brokerURL" value="tcp://localhost:61616
&jms.redeliveryPolicy.maximumRedeliveries=10
&jms.redeliveryPolicy.redeliveryDelay=5000\"/>
</bean>
```

Each of the redelivery options must be prefixed with `jms.redeliveryPolicy..` In the example above we configured maximum redeliveries to ten, and use five 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 redelivery so that message ordering is maintained while messages appears as inflight on the broker. That means redelivery of messages from the given queue is limited to a single consumer, as the broker is essentially unaware that the consumer is performing redeliveries. If message ordering is not important and/or higher throughput and load distribution among concurrent consumers is desired, then consumer level redelivery is not sufficient and broker level should be used instead.

CONFIGURING BROKER LEVEL REDELIVERY WITH ACTIVEMQ

Broker level redelivery is configured as an ActiveMQ plugin which is 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 the example above we perform at most 10 redeliveries with ten seconds delay in between, however the first delay is only five seconds. You can find more details about ActiveMQ redelivery at the ActiveMQ web site. We will now talk about redeliveries when we start from a database.

TRANSACTION REDELIVERIES STARTING FROM DATABASE

As opposed to a message broker, a database does not have the concept of redelivery or dead letter queue, therefore you have limited options what you can do to address repeated transaction rollbacks when starting from a database.

The source code for the book contains an example that starts a transaction from a database in the chapter12/tx-database directory. There is a rollback example which you can try using the following Maven goal:

```
mvn test -Dtest=SpringRollbackTest
```

The SQL consumer will by default poll the database every half second, and because the example is designed to fail, it will 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 that may detect a failure every half a second. In those situations you can configure the SQL consumer to back off polling so frequently as shown below:

```
<from uri="sql:{{sql-from}}?consumer.onConsume={{sql-delete}}
&dataSource=#myDataSource&transacted=true
&backoffMultiplier=5&backoffErrorThreshold=1"/>
```

The options `backoffErrorThreshold=1` and `backoffMultiplier=5` tells Camel to multiple the polling interval with 5 after there has been one error, so the polling will happen every $0.5 * 5 = 2.5$ seconds. You want to use this in these situations where 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 a second again.

TIP The SQL endpoint also provides a back off capability when there is no data, using the `backoffIdleThreshold` option. For example `backoffIdleThreshold=3` will trigger backoff after 3 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 there can be situations where you need more fine-grained control, such as when specifying transaction propagation settings.

12.3.5 Transaction propagations

When you configure transactions, you'll come across the term *transaction propagation*. In this section, you'll learn what that term means and why it's related to configuring transactions.

If you have ever worked with Enterprise Java Beans (EJBs), you may be familiar with transaction propagation already. Transaction propagation options specify what will happen if a method is invoked and a transaction context already exists. For example, should it join the existing transaction? Should a new transaction be started? Or should it fail?

In most use cases, you end up using one of two options: joining the existing transaction (`PROPAGATION_REQUIRED`) or starting a new transaction (`PROPAGATION_REQUIRE_NEW`).

To use transaction propagation, you must configure it in the XML file as shown in the listing 12.6.

Listing 12.6 Configuring and using transaction propagation

```

<bean id="required"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager"/>
    <property name="propagationBehaviorName"
              value="PROPAGATION_REQUIRED"/>
</bean>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route id="partnerToDB">
        <from uri="activemq:queue:partners"/>
        <transacted ref="required"/>          ③
        <bean ref="partner" method="toSql"/>
        <to uri="jdbc:myDataSource"/>
    </route>
</camelContext>

```

- ① <bean> defining policy with propagation required
- ② Specifying the propagation behavior as required
- ③ Refer to the configured policy

First you define a bean with the id "required", which is a `SpringTransactionPolicy` type ① . The bean must refer to both the transaction manager and the choice of transaction propagation to be used ② . In the Camel route, you then refer to the required bean from within the `<transacted>` tag, using the `ref` attribute ③ .

If you want to use `PROPAGATION_REQUIRE_NEW`, it's just a matter of changing the property on the bean as shown in bold:

```

<property name="propagationBehaviorName"
          value="PROPAGATION_REQUIRE_NEW"/>

```

You may ask yourself what kinds of transaction propagation policies exists, well table 12.1 answers the question.

Table 12.1 - List of possible transaction propagation policies

| Transaction Propagation Policy | Description |
|--------------------------------|---|
| PROPAGATION_MANDATORY | Support a current transaction; throw an exception if no current transaction exists. |
| PROPAGATION_NESTED | Execute within a nested transaction if a current transaction exists, behave like PROPAGATION_REQUIRED else. |
| PROPAGATION_NEVER | Do not support a current transaction; throw an exception if a current transaction exists. |
| PROPAGATION_NOT_SUPPORTED | Do not support a current transaction; rather always execute non-transactionally. |
| PROPAGATION_REQUIRED | Support a current transaction; create a new one if none exists. |

| | |
|-------------------------|---|
| PROPAGATION_REQUIRE_NEW | Create a new transaction, suspending the current transaction if one exists. |
| PROPAGATION_SUPPORTS | Support a current transaction; execute non-transactionally if none exists. |

You have now learned how to configure and use transactions with Camel. But there's more to learn. In the next section, we'll look at how transactions work when you have multiple routes and the following section when you need different propagation behavior.

12.3.6 Using transactions with multiple routes

In Camel, it's common to have multiple routes and to let one route reuse another by sending messages to it. In this section, we'll look at how this works when one or all routes are transacted. Then we'll look at some of the implications of using transactions with request-response messaging style.

We'll start out simply and look at what happens when you use a non-transacted route from a transacted route.

USING TRANSACTIONS WITH A NON-TRANSACTIONED ROUTE

Listing 12.7 shows the parts of a unit test that you can use to see what happens when a transacted route calls a non-transacted route.

Listing 12.7 Unit test with a transacted route calling a non-transacted route

```
public class TXTToNonTXTTest extends CamelSpringTestSupport {

    protected AbstractXmlApplicationContext createApplicationContext() {
        return new ClassPathXmlApplicationContext("spring-context.xml"); ①
    }

    protected RouteBuilder createRouteBuilder() throws Exception {
        return new SpringRouteBuilder() {
            public void configure() throws Exception {
                from("activemq:queue:a")
                    .transacted() ②
                    .to("direct:quote")
                    .to("activemq:queue:b");

                from("direct:quote")
                    .choice() ③
                        .when(body().contains("Camel"))
                            .transform(constant("Camel rocks"))
                        .when(body().contains("Donkey"))
                            .throwException(new IllegalArgumentException(
                                "Donkeys not allowed"))
                    .otherwise()
                        .transform(body().prepend("Hello "));
            }
        };
    }
}
```

① The Spring XML file to use

- ② Transacted route
- ③ Non-transacted route

In listing 12.7, you first import the Spring XML file ①, which contains all the Spring configuration to set up the JMS broker, Spring, and the Camel ActiveMQ component. The content of the `spring-context.xml` file is the same as in listing 12.4. Next, you have the two routes. First, the transacted route ③ moves messages from queue A to B. During this move, the message is also processed by the non-transacted route ④, which transforms the message using a content-based router. Notice that if the message contains the word "Donkey", the route will force a failure by throwing an exception.

You can run this unit test by running the following Maven goal from the chapter9/multiple-routes directory:

```
mvn test -Dtest=TXToNonTXTest
mvn test -Dtest=SpringTXToNonTXTest
```

The unit test has three methods: two test situations that commit the transaction, and one rolls back the transaction because of the exception being thrown. Here are two tests showing the commit and rollback situations:

```
public void testWithCamel() throws Exception {
    template.sendBody("activemq:queue:a", "Hi Camel");
    Object reply = consumer.receiveBody("activemq:queue:b", 10000);
    assertEquals("Camel rocks", reply);
}

public void testWithDonkey() throws Exception {
    template.sendBody("activemq:queue:a", "Donkey");
    Object reply = consumer.receiveBody("activemq:queue:b", 10000);
    assertNull("There should be no reply", reply);
    reply = consumer.receiveBody("activemq:queue:ActiveMQ.DLQ", 10000);
    assertNotNull("It should have been moved to DLQ", reply);
}
```

What can you learn from this? The unit test proves that when a transacted route uses a non-transacted route, the transactional behavior works as if all routes are transacted, which is what you'd expect. The last unit test proves that when the non-transacted route fails by throwing an exception, the transacted route detected this and issued a rollback. You can see this because the message is moved to the JMS broker's dead letter queue.

This is great news, because there are no surprises. It's safe for transacted routes to reuse existing non-transacted routes.

NOTE The transaction manager requires messages to be processed in the *same* thread context, to support the transaction. This means that when you use multiple routes, you must link them together in a way that ensures the message is processed in the same thread. Using the Direct component does this—the Direct component was used in listing 12.7 to link the two routes together. This won't work with the SEDA component, which routes messages using another thread.

Let's continue and see what happens when both routes are transacted.

USING TRANSACTIONS WITH ANOTHER TRANSACTED ROUTE

Now let's modify the unit test from listing 12.8 and create a new situation, where both routes are transacted, and see what happens. Here are the routes.

Listing 12.8 Two transacted routes

```
public void configure() throws Exception {
    from("activemq:queue:a")
        .transacted()
        .to("direct:quote")
        .to("activemq:queue:b");

    from("direct:quote")
        .transacted()
        .choice()
            .when(body().contains("Camel"))
                .to("activemq:queue:camel")
            .otherwise()
                .throwException(new IllegalArgumentException
                    ("Unsupported animal"));
}
```

You can run this example by running the following Maven goal:

```
mvn test -Dtest=TxtToTxtTest
mvn test -Dtest=SpringTxtToTxtTest
```

Once again, the unit test will prove that there are no surprises here. When the exception is thrown, the entire route is rolled back, which is what you'd expect. When the message hits the second route and the second `transacted`, it participates in the existing transaction. This is because `PROPAGATION_REQUIRED` is the default propagation behavior when using `transacted`.

Next, we'll make it more challenging by using two different transaction propagations.

12.3.7 Using different transaction propagations

In some situations, you may need to use multiple transactions with the same exchange, as illustrated in figure 12.10.

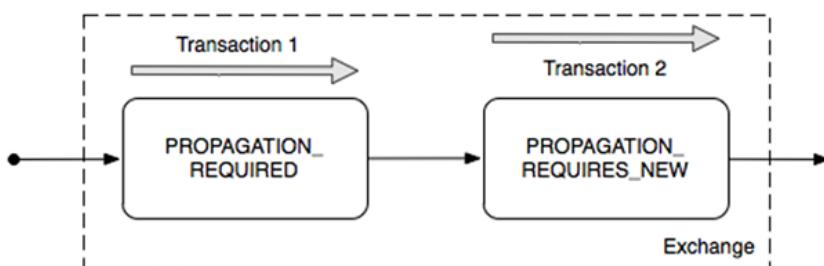


Figure 12.11 Using two independent transactions in a single Exchange

In figure 12.10 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_REQUIRED`, which will start a new transaction regardless of whether an existing transaction exists or not. 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 there are two transaction legs in play, then they can have different outcome, for example transaction 1 can rollback, while transaction 2 will commit, and vice-versa.

In Camel, a route can only be configured to use at most one transaction propagation, which means figure 12.10 must use two routes. The first route uses `PROPAGATION_REQUIRED` and the second route uses `PROPAGATION_REQUIRE_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 listing 12.9.

Listing 12.9 - 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"/>
    <to uri="direct:audit"/>
    <to uri="direct:order"/>
    <to uri="activemq:queue:order"/>
  </route>

  <route>
    <from uri="direct:audit"/>
    <transacted ref="requiresNew"/>
    <bean ref="auditLogService" method="insertAuditLog"/>
  </route>

  <route>
    <from uri="direct:order"/>
    <transacted ref="mandatory"/>
    <bean ref="orderService" method="insertOrder"/>
  </route>
</camelContext>
```

- ① Require transaction
- ② Call the audit-log route
- ③ Call the insert-order route
- ④ Use a new transaction for the audit-log route
- ⑤ Use the existing parent transaction for the insert order route

The first route uses `PROPAGATION_REQUIRED` to start transaction-1 ① . Then the audit-log route is called ② which uses `PROPAGATION_REQUIRE_NEW` ④ to start transaction-2 that runs

independently from transaction-1. After the message has been inserted into the audit-log, then the insert-order route is called ③ that will reuse the existing transaction-1 from the parent route ⑤ which is specified by using the mandatory ⑥ propagation property.

In listing 12.9 we are using three different propagation behaviors which are configured as shown in listing 12.10.

Listing 12.10 Configure three different 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_REQUIRE_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 source code for the book 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
```

There are four unit tests which demonstrate different scenarios as described in table 12.2.

Table 12.2 Overview of what happens when transaction 1 or 2 either commit or rollback

| Test Method | Transaction-1 | Transaction-2 | Comment |
|----------------|---------------|---------------|---|
| testWithCamel | commit | commit | The message is successfully processed. In the database one row is inserted in both the order and audit-log table. |
| testWithDonkey | rollback | commit | The message fails during order validation, and is rolled back. In the database no rows are 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 audit-logs even if transaction-1 fails. |

| | | | |
|---------------------------|----------|----------|--|
| testAuditLogFail | rollback | rollback | The message fails during audit logging, and is rolled back. In the database no rows are inserted as both transactions rollback. |
| testAuditLog-RollbackLast | commit | rollback | The message fails during audit logging, and only the 2nd transaction is explicitly marked to rollback, leaving the 1st transaction to commit. This means only the order is inserted into the database, whereas the audit-log failed and was rolled back. |

Keep it simple

Despite you can use multiple propagation behaviors with multiple routes in Camel, then use this with care. Try to design your solutions with as few propagations as possible, as complexity arises dramatically when you introduce new propagation behaviors. For example table 12.2 lists four outcomes you have using two propagations, if you have three propagations then 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?

12.3.8 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 12.11 illustrates the application.

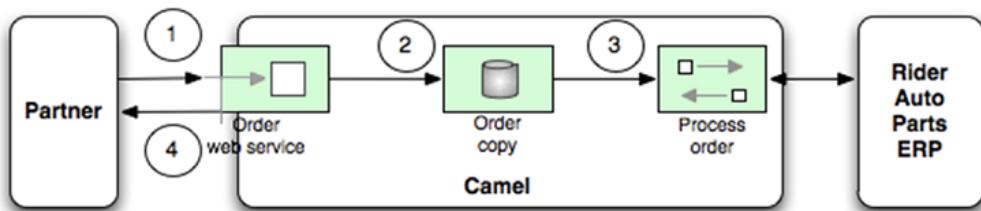


Figure 12.12 A web service used by business partners to submit orders. A copy of the order is stored in a database before it's processed by the ERP system

As you can see in the figure, 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 leverage it with their IT systems. There is a single return code that indicates whether or not 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. This means that 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 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 leverage `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 rollback the transaction you mark it for rollback using the 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 the `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

There are several ways you can rollback a transaction. By default if any unhandled exception happens during routing the transaction error handler detects this and stops continue routing. The caused exception is then propagated back to the transaction manager, that marks the transaction for rollback. In other words unhandled exceptions triggers a transactional rollback. In the Camel routes you can make rollbacks obvious 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 source code for the book contains this example in the `chapter12/riderautoparts-order` directory. You can start the application by 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 (<http://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 12.12. You then send the request by clicking the green play button, and it will display the reply in the pane on the right side.

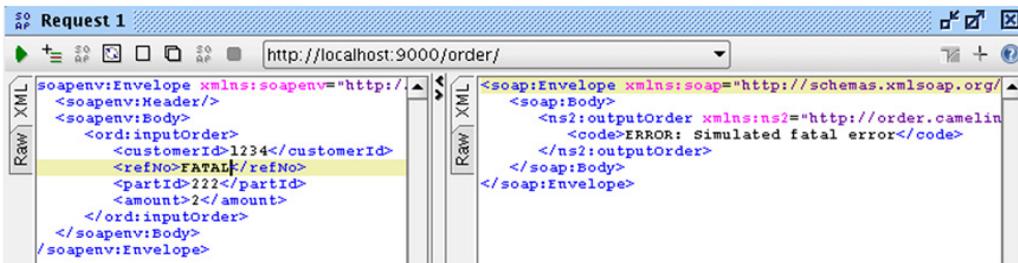


Figure 12.13 A web service message causes the transaction to roll back, and a custom reply message is returned.

Figure 12.12 shows an example where 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 12.12 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. So what can you do instead? In the next section, we'll look 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, we'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 that a transaction manager does. 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 the 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 or not.

Figure 12.13 illustrates how these concepts are related to each other. As you can see from this figure, 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 is not shared with others.

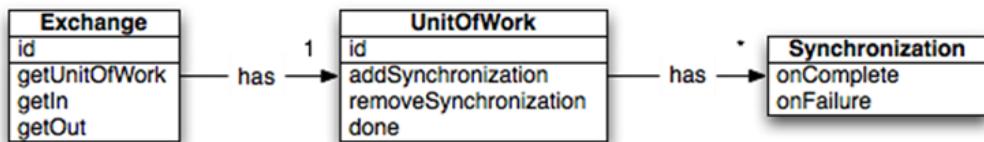


Figure 12.14 An `Exchange` has one `UnitOfWork`, which in turn has from zero to many `Synchronizations`.

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 leverage 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. Don't worry about throwing exceptions from your custom `Synchronization`—Camel will catch those and log them at `WARN` level, and will then continue to invoke the next callback. This ensures that all callbacks are invoked even if one happens to fail.

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 12.14 shows the principle of this application.

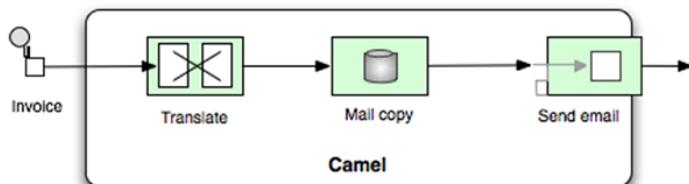


Figure 12.15 Emails are sent to customers listing their invoice details. Before the email is sent, a backup is stored in the file system.

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 only override either 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 12.14. This can be done using the Java DSL as highlighted:

```
public void configure() throws Exception {
    from("direct:confirm")
        .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                exchange.getUnitOfWork()
                    .addSynchronization(new FileRollback());  
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}");
}
```

① Inlined Processor

The source code for the book contains this example in the chapter12/uow directory. You can try it by 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 exchangeId:
  9edc1ecb-43be-43ee-9f32-7371452967bd.
WARN FileRollback - Failure occurred so deleting backup file:
  target/mail/backup/02630ec4-724d-4e73-8eb6-c969720578c
```

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 ① using lambda style, which reduces the code from six lines to only one line as shown:

```
.process(e -> e.addOnCompletion(new FileRollback()))
```

But it still requires the inlined Processor ① with our without Java 8 lambda style. Isn't there a more convenient way? Yes there is, 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`.

So let's see how it's done. `OnCompletion` is available in both Java DSL and XML DSL variations. Here's how `onCompletion` is used with XML DSL.

Listing 12.11 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">
        <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>
```

①

① Context scoped `onCompletion` that triggers on failure only

As you can see from the code, `<onCompletion>` is defined as a separate Camel route. The `OnCompletion` will be executed right after the message is doing being routed. In the example above, that means after the last step in the route:

```
<log message="Mail sent to ${header.to}"/>
```

.. the message reached the end and the `onCompletion` is executed.

Under the hood

When you add an `<onCompletion>` to a `<camelContext>` or `<route>` the `<onCompletion>` will be added as a sub route that is being routed from a `Synchronization` callback. The routing engine is responsible for orchestrating this, by detecting whether an `<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, then the `UnitOfWork` kicks in and calls each `Synchronization` callback, and hence among those the callback that triggers the `<onCompletion>` sub route.

In the present situation you're only interested in executing `onCompletion` when the exchange fails, so you can specify this by setting the `onFailureOnly` attribute to `true`.

The source code for the book contains this example in the chapter12/uow directory, which you can run using the following Maven goal:

```
mvn test -Dtest=FileRollbackTest
mvn test -Dtest=SpringFileRollbackTest
```

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 in situations where the exchange did not 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 `onException` does at either context or route scope (as you saw in chapter 11). You could create a Java DSL-based version of listing 12.11 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 only trigger when the predicate matches.

USING PREDICATE WITH ONCOMPLETION

Suppose we only want the `OnCompletion` to trigger if a file was saved. To do this we 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 we 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 we want to run `OnCompletion` before the consumer returns a response to the caller we have to switch mode

USING BEFORECONSUMER MODE

As part of your work for Rider Auto Parts you recently begun 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 12.15 illustrates this principle.

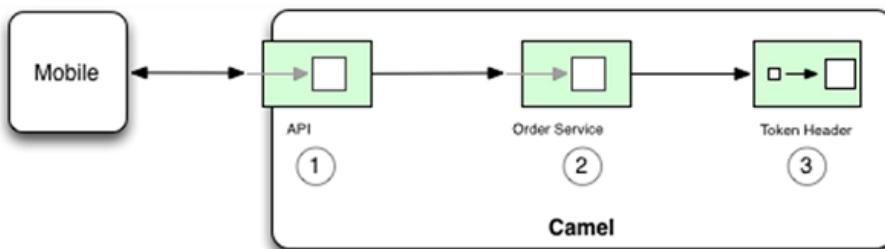


Figure 12.16 A mobile accesses a Camel API that fetches order information and enriches the response with a token header

Figure 12.15 shows how mobile user 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 is valid, then the response must be enriched with a token ③ that the mobile client must use for subsequent queries to the API.

The API is implemented using Camel in a `RouteBuilder` class using the `rest-dsl` as shown in listing 12.12.

Listing 12.12 - API implemented using rest-dsl in Camel

```

public void configure() throws Exception {
    restConfiguration().component("netty4-http")
        .bindingMode(RestBindingMode.json)
        .host("localhost").port(8080).contextPath("service");
    onCompletion().modeBeforeConsumer()
        .setHeader("Token").method(TokenService.class);
    rest()
        .get("/order/{id}")
        .to("bean:orderService?method=getOrder");
}
    
```

- ① Configure rest on localhost:8080/service using json binding
- ② rest-dsl defining a service to get order by id

③ OnCompletion that adds the token header to be included in the response

At first you setup the rest configuration to use the netty4-http component as the HTTP server

① . You are using Netty because it is a very popular and fast and lightweight networking framework; which works well in these times of micro services. The REST services is exposes on localhost:8080/service and using json binding mode.

In the 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 is now a requirement that a token header is returned with an unique value. The mobile service is then required to use the token header for subsequent access to the API when other services is requested. To add the unique token header you use OnCompletion ③ . Switching to BeforeConsumer mode, ensures the header is added to the message before the rest service returns the response to the mobile client.

The source code of the book 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, then 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, as shown below:

```
davsclaus:~/ $ curl -i http://localhost:8080/service/order/123
HTTP/1.1 200 OK
Content-Length: 37
Accept: /*
breadcrumbId: ID-davsclaus-air-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 then notice 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 have talked about how transactions, can be used as a mean to coordinate state updates among distributed systems to form a unit of work as a whole.

Related to this concept is idempotency which is 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 the term

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. For example when a sender sends a message to a receiver ①, the sender can only know the receiver have received and accepted the message, if an acknowledge message is returned from the receiver. However if that acknowledgement is lost due to networking issue ②, then the sender may need to resend the message, that the receiver have already received - a duplicate message ③. Figure 12.16 illustrates this principle.

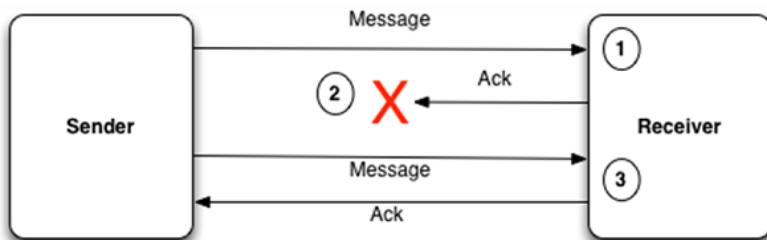


Figure 12.17 The receiver receives duplicate message because of problem of returning acknowledgement from the receiver to the sender

Therefore many messaging systems, such as Apache ActiveMQ, have capabilities to eliminate duplicate messages, so the users do not 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:

- Generate unique key for each message
- Store and retrieve previously seen keys

The heart of the idempotent consumer implementation is the repository which is defined as an 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);
  
```

How this works in Camel is illustrated in figure 12.17 and explained in the following steps:

1. When an Exchange reaches the idempotent consumer, the unique key is evaluated from the Exchange using the configured Camel Expression. For example this can be a message header, or an XPath expression etc.
2. The key is checked against the idempotent repository whether its a duplicate or not. This is done by calling the `add` method. If the method returns `false`, then the key was successfully added and no duplicate as detected. If the method returns `true`, then an existing key already exists and the message is detected as a duplicate.
3. If the Exchange is not a duplicate then its routed as usual.
4. When the Exchange is done being routed, then its Synchronization callbacks is invoked whether to commit or rollback the Exchange.
5. To commit the Exchange the `confirm` method is invoked on the idempotent repository.
6. To rollback 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 the same message, then Camel will have no prior knowledge of having seen this message before, and therefore not 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. What would happen instead is that upon rollback the key is not removed but confirmed instead (upon rollback the same action is performed as a commit - step 5)

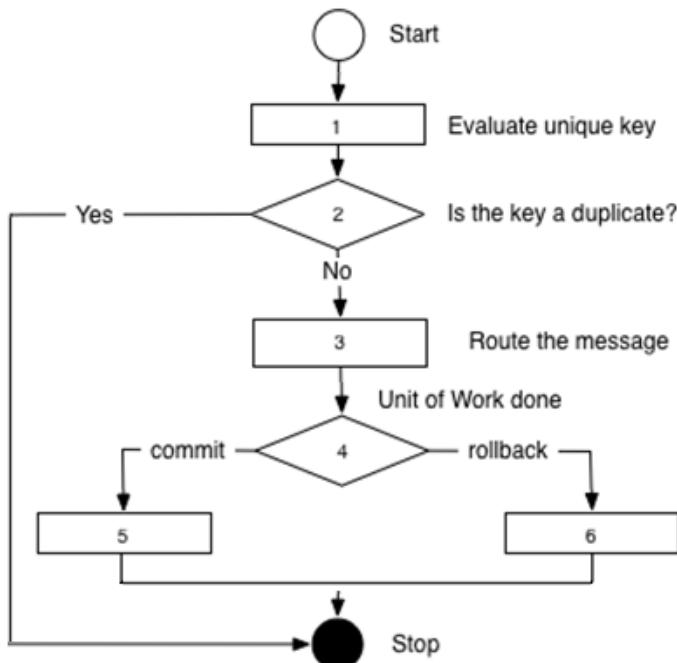


Figure 12.18 How the idempotent consumer EIP operates in Camel

Lets see how to do this in action.

USING THE IDEMPOTENT CONSUMER EIP

In order to use the idempotent consumer EIP, we need to setup the idempotent repository first. Listing 12.13 shows an example how to do this using XML DSL.

Listing 12.13 - Using Idempotent Consumer EIP in XML DSL

```
<bean id="repo" ①
      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
            <header>orderId</header>
            <log message="Processing order ${header.orderId}"/>
            <to uri="mock:order"/>
        </idempotentConsumer>
    </route>
</camelContext>
```

- ① Using the memory idempotent repository
- ② Route segment using the idempotent consumer
- ③ Expression to evaluate the unique key

In this example we use the built-in memory based repository, which is defined as a bean in XML DSL ① . In the Camel route we then wrap the routing segment that we want to only be invoked once per unique message ② . The attribute messageIdRepositoryRef ② must be configured to refer to the chosen repository ① . The example using the message header orderId as the unique key ③ .

NOTE The unique key is evaluated as a String type. Therefore you can only use information from the Exchange which can be converted to a String type. In other words you cannot use complex types such as Java objects unless they generate unique keys using their `toString` method.

The example from listing 12.13 can be done as following using Java DSL.

Listing 12.14 - Idempotent Consumer using Java DSL

```
public void configure() throws Exception {

    IdempotentRepository repo = new MemoryIdempotentRepository() ①

    from("seda:inbox")
        .log("Incoming order ${header.orderId}")
        .to("mock:inbox")
```

```

    .idempotentConsumer(header("orderId"), repo)
        .log("Processing order ${header.orderId}")
        .to("mock:order")
    .end();
}

```

- ① Using the in-memory idempotent repository
- ② Route segment using the idempotent consumer, and expression to evaluate unique key

The source code of the book contains this example in the chapter12/idempotent directory, which you can try using the following Maven goals:

```
mvn test -Dtest=SpringIdempotentTest
mvn test -Dtest=IdempotentTest
```

Running this example will send 5 messages to the Camel route of which 2 messages will be duplicated, so only 3 messages will be processed by the idempotent consumer. The example uses the following code to send the 5 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");
```

Which outputs the following to the console:

```
2016-04-12 [ - seda://inbox] INFO  route1 - Incoming order 123
2016-04-12 [ - seda://inbox] INFO  route1 - Processing order 123
2016-04-12 [ - seda://inbox] INFO  route1 - Incoming order 123
2016-04-12 [ - seda://inbox] INFO  route1 - Incoming order 789
2016-04-12 [ - seda://inbox] INFO  route1 - Processing order 789
2016-04-12 [ - seda://inbox] INFO  route1 - Incoming order 456
2016-04-12 [ - seda://inbox] INFO  route1 - Processing order 456
2016-04-12 [ - seda://inbox] INFO  route1 - Incoming order 789
```

As you can see the orders are only processed 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 a great number of idempotent repositories, each supporting different level of service - some are for standalone in memory only, others support clustered environments. Table 12.3 list all the current implementations provided by Apache Camel 2.19 with some tips about pros and cons.

Table 12.3 IdempotentRepository implementations offered by Apache Camel 2.19

| Idempotent Repository | Description |
|------------------------------------|--|
| MemoryIdempotentRepository | A very fast in-memory store that stores entries in a Map structure. All data will be lost when the JVM is stopped. 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 is synchronized which means the store can be a potential bottleneck for very high concurrent throughput. Each write operation re-writes the file which means for very large data sets it can be slower to write. 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 an unique name, but can reuse the same database table to store keys. High volume processing can become a performance bottleneck if the database cannot 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 its required to setup the needed SQL table using the SQL script listed on the camel-sql documentation webpage. |
| JpaMessageIdRepository | 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 auto create needed SQL tables. |
| MongoDbIdempotentRepository | A store using MongoDB as the database. This store is provided in the camel-mongodb module. |
| NamedCassandraIdempotentRepository | A store using Apache Cassandra as the database. This implementation uses Cassandra CQL queries. Clustering is supported as that is native supported by Cassandra. This store is provided in the camel-cassandraql module. To use this store its required to setup the needed table using the CQL script listed on the camel-cassandra documentation webpage. |
| HazelcastIdempotentRepository | 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 also write the Map structure to persistent store using different levels of QoS. This implementation is suitable for clustering. This store is provided in the camel-hazelcast module. |
| InfinispanIdempotentRepository | Similar to Hazelcast but using JBoss Infinispan instead as the in-memory data grid. This store is provided in the camel-infinispan module. |
| JCacheldempotentRepository | Using the JCache API which allows to plugin different JCache providers such as Hazelcast and Infinispan. This stores is provided in the camel-jcache module. |

| | |
|----------------------------------|---|
| HBaseIdempotentRepository | A store using Apache HBase as the database. This store should only be considered in use if users already have an existing Hadoop installation. As it would be overkill to setup Hadoop to only be used by Camel as the idempotent repository. This implementation is suitable for clustering. This store is provided in the camel-hbase module. |
| RedisIdempotentRepository | A store using Redis key/value store. This implementation is suitable for clustering. This store is provided in the camel-spring-redis module. |

It is also possible to write your own custom implementation, by just implementing the `org.apache.camel.spi.IdempotentRepository`. If you are looking for examples how to do that, then remember each of the implementations listed in table 12.2 can be used as inspiration.

Cache eviction

Each implementation of the idempotent repository, listed in table 12.2, has different strategy for cache eviction. The in-memory based has a limited size with the number of entries they keep in the cache using a LRU (Least Recently Used) cache. The size of this cache can be configured. When there are more keys to be added to the cache, then the least used is discarded first.

Other implementations such as SQL and JPA requires you to manually delete unwanted records from the database table.

In memory data grids such as Hazelcast, Infinispan and JCache provides configuration for setting the eviction strategy.

It is recommended to study the documentation of the used technology to ensure you use the proper eviction strategy.

To solve a common use case with Camel that allows multiple Camel applications to process files from a shared directory, involves using clustered idempotent repository, to implement a working solution.

12.5.3 Clustered idempotent repository

This section covers a common use cases involving clustering and the idempotent consumer pattern. In this use case a shared file system is used for exchanging data, which you want to process as fast as possible by scaling up your system in a cluster of active nodes, where each node can pickup incoming files and process them.

The file component from Apache Camel has built-in support for idempotency. But due to historical reasons this implementation does not support atomic operations a clustered solution would require; there is a small window of opportunity that duplicates could still happen. Figure 12.18 illustrates this principle.

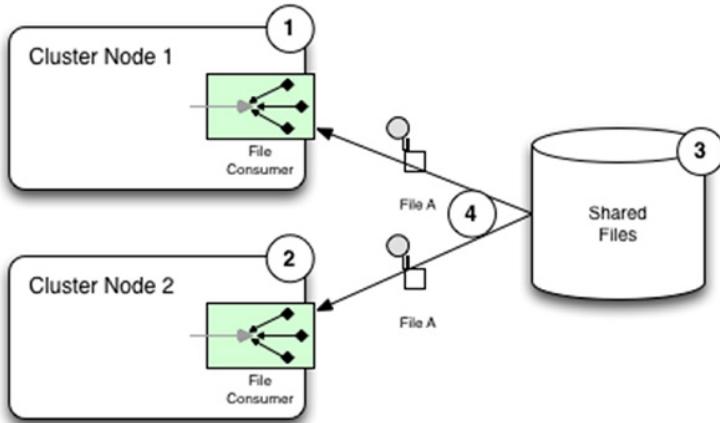


Figure 12.19 Two clustered nodes (1)(2) with the same Camel application competes concurrently to pickup new files from a shared file system (3) and they may pickup the same file (4) which causes duplicates.

When a new file is written to the shared file system (3), then each active node (1)(2) in the cluster reacts independently and depending on timing and non deterministic factors, then the same file can potentially be picked up by one or more nodes (4). This is not what we want to happen. We want only one node to process a given file at any time, but we also want each node to process different files at the same time, to archive higher throughput and distribute the work across the nodes in the cluster, so node 1 can process file A while node 2 process file B and so forth.

What we need is something stronger ... yeah maybe take a break after reading so far, and grab a strong drink, or well if you are in the office, then a cup of coffee or tea will do just fine.

The solution we are looking for is a clustered idempotent repository, that the nodes uses to coordinate and grant locks to exactly only one node. Figure 12.19 shows how this plays out in Camel.

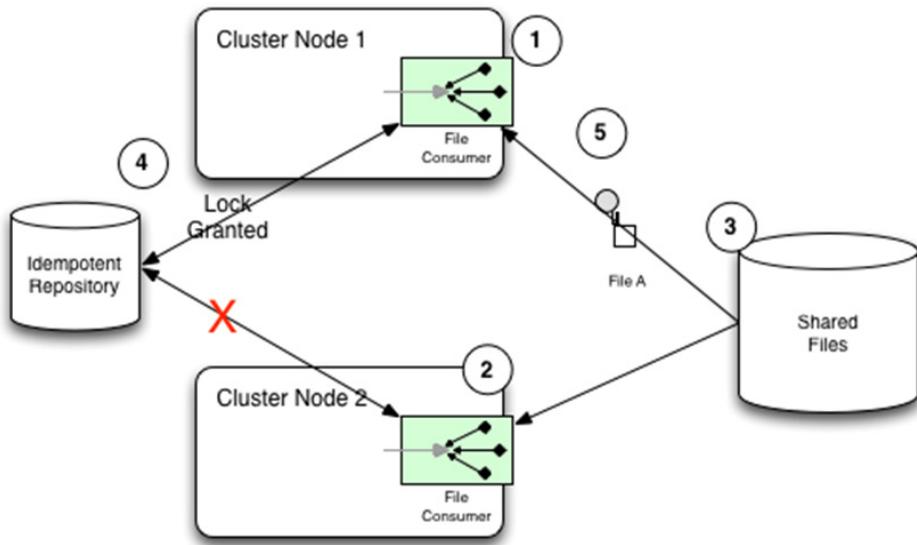


Figure 12.0 Two clustered nodes ① ② with the same Camel application competes concurrently to pickup new files from a shared file system ③ . Both nodes tries to acquire a read lock ④ that will permit exactly one with permission to pickup and process the file ⑤ . The other nodes will fail to acquire the lock and will skip attempting to pickup and process the file.

Each node in the cluster ① ② as before reacts when new files are available from the shared file system ③ . When a node detects a file to pickup, it now first attempts to grab an exclusive read lock ④ from the clustered idempotent repository. If granted then the file consumer can pickup and process the file ⑤ . And when its done with the file, the read lock is released. While the read lock is granted to a node, then any other node that attempts to acquire a read lock for the same file name, would be denied by the clustered idempotent repository. The read lock is per file, so each node can different files concurrently.

Lets see how we can 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 we first need to configure Hazelcast which we can either use as a client to an existing external Hazelcast cluster, or we can embed Hazelcast together with our Camel application.

Hazelcast can be configured using Java code or from a XML file. We use the latter in this example and have copied the sample config that is distributed from Hazelcast and located the file in `src/main/resources/hazelcast.xml`. Hazelcast allows each node to auto join using multicast. By doing this we do not 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 runs out of the box on personal computers, as not all popular operating systems have UDP enabled out of the box. We have left code comments in the `hazelcast.xml` file how we did that.

The source code for the book contains this example in the `chapter12/hazelcast` directory. To represent two nodes we have two almost identical source files `camelinaction.ServerFoo` and `camelinaction.ServerBar`. Listing 12.15 shows the source code for `ServerFoo`.

Listing 12.15 - A standalone Java Camel application with Hazelcast embedded

```
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import org.apache.camel.main.Main;
import org.apache.camel.processor.idempotent.hazelcast.HazelcastIdempotentRepository;

public class ServerFoo {

    private Main main;

    public static void main(String[] args) throws Exception {
        ServerFoo foo = new ServerFoo();
        foo.boot(); ①
    }

    public void boot() throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance(); ②

        HazelcastIdempotentRepository repo = new HazelcastIdempotentRepository(hz, "camel"); ③

        main = new Main();
        main.enableHangupSupport();
        main.bind("myRepo", repo); ④
        main.addRouteBuilder(new FileConsumerRoute("FOO", 100));
        main.run(); ⑤
    }
}
```

- ① `ServerFoo` is a standalone Java application with a `main` method to boot the application.
- ② Create and embed a Hazelcast instance in server mode
- ③ Setup Camel Hazelcast Idempotent Repository using the name `camel`.
- ④ Bind the Idempotent Repository to the Camel registry with the name `myRepo`
- ⑤ Add a Camel route from Java code and run the application

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 we 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 being started up, and therefore ready to use the Hazelcast idempotent repository is created ③ . The remainder of the code is using 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 listing 12.16.

Listing 12.16 - 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"
            + "&idempotentRepository=#myRepo")
            .log(name + " - Received file: ${file:name}")
            .delay(delay)
            .log(name + " - Done file: ${file:name}")
            .to("file:target/outbox");
    }
}
```

- ➊ The read lock must be configured as idempotent
- ➋ The idempotent repository to use
- ➌ Delays processing the file so we humans has a chance to see what happens

The Camel route is simply a file consumer that pickup files from the target/inbox directory (which is the shared directory in this example). The consumer is configured to use idempotent ➊ as its read lock. The idempotent repository is configured with the value #myRepo ➋ which was the name we used in listing 12.15. When a file is being processed its logged and delay for a little bit, so the application runs a bit slower, so we can better see what happens.

You can try this example 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 its a standard Java main application - the IDE typically supports this having a right click menu to run Java applications).

```
mvn compile exec:java -Pfoo
mvn compile exec:java -Pbar
```

When you start the 2nd application then Hazelcast should log the cluster state as shown below:

```
Members [2] {
    Member [localhost]:5701
    Member [localhost]:5702 this
}
```

Here we can see there are two members in the clustered, and that they are linked together using TCP using ports 5701 and 5702.

If you copy a bunch of files to the target/inbox directory, then each node will compete concurrently to pickup and process the files. When a node cannot acquire an exclusive read lock a WARN is logged as shown:

```
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 is not a WARN from the other node, and vice-versa.

We can reduce the number of read-lock contention by shuffling the files by random order. This can be done by configuring the file endpoint with `shuffle=true`. For example the when the author of this chapter tried this then 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.

EVICTION STRATEGY

Hazelcast will by default keep each element in its distributed cache forever. As this is often not what you want, as you may want to be able to pickup a file in the future having the same file as a previously processed file, or if you are using unique file names, then keeping old files in the cache is a waste of memory. To remove these old file names from the cache, an eviction strategy must be configured. Hazelcast has many options for this, and in this example we have 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. By leveraging 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 only be used by a limited number of resources, such as JMS and JDBC. Therefore, it only makes sense to use transactions when you can leverage 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 only involves one transactional resource, use local transactions. They're simpler and much less resource intensive. Only use global transactions if multiple resources are involved.
- *Configuring global transactions is difficult.* When using global transaction (JTA/XA) then be advised that it's difficult to configure. The configuration is vendor and application

server specific. Do not underestimate how difficult it can be.

- *Short transactions.* Avoid building systems where transactions spans across a lot of business logic and being routed to many other systems. In other words 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 is 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 are in need for custom logic to be executed as either commit or rollback then 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 to understand and use the most appropriate for your use case.
- *Clustered file consumer.* Make sure to use idempotent read lock with a clustered idempotent repository such as Hazelcast, Infinispan or JCache when you want to scale out your application and have multiple Camel applications process files from a shared directory.

We'll now 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
- 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 waits 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 very 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 leverages the concurrency features from Java, so we'll first discuss how concurrency works in Java before we can move 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 how you can use concurrency with the EIPs.

The last section of this chapter focuses on how you can achieve high scalability with Camel and how you can leverage 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 at

first to understand and do the right way. However it's the price to pay for having Camel achieve high scalability (routing using non-blocking mode).

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, IO-bound, database-bound, and so on. Integration applications are often IO-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 by utilizing 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. This means there's often a mix of resources, which are either CPU-bound or IO-bound. It's very likely you can achieve higher performance by 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 their 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 are therefore based on what would be considered legacy today. The data is exchanged between the stakeholders using files that are transferred using FTP servers.

Figure 13.1 illustrates this business process.

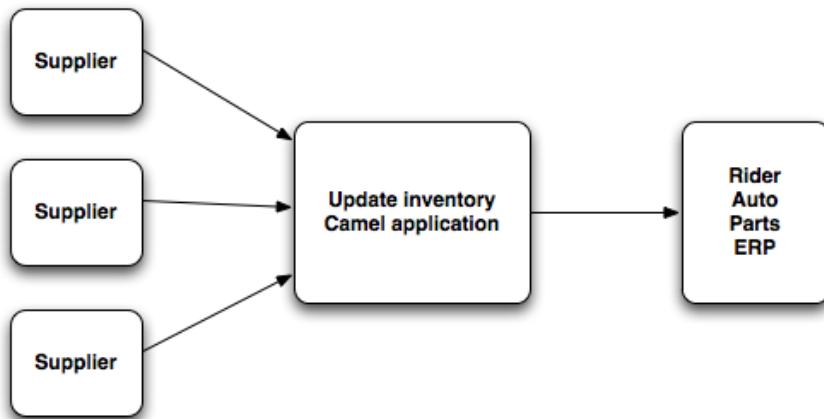


Figure 13.1 Suppliers send inventory updates, which are picked up by a Camel application. The application synchronizes the updates to the ERP system.

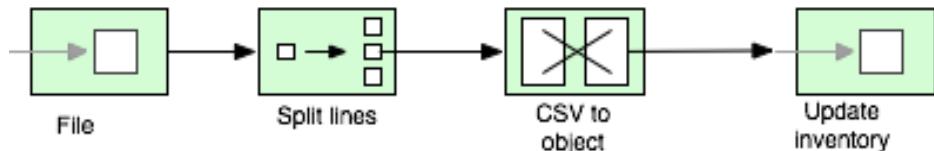


Figure 13.2 A route picks up incoming files, which are split and transformed to be ready for updating the inventory in the ERP system.

Figure 13.2 shows the route of the inventory-updating Camel application from figure 13.1. 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.

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()
            .bean(InventoryService.class, "csvToObject")
            .to("direct:update")
        .end()
        .log("Done processing file: ${header.CamelFileName}");

    from("direct:update")
        .bean(InventoryService.class, "updateInventory");
}

```

- ➊ Splits file line by line
- ➋ Updates ERP system

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 ➊. This is done by using the Splitter EIP in *streaming mode*. The 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 ➋.

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 different techniques for speeding things up by leveraging concurrency. But first we'll 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 source code for the book 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 1000 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 the bigfile.csv without concurrency. This is done by using the following Maven goal:

```
mvn test -Dtest=BigFileTest
```

When the test runs, it will 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. 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, we'll see three different solutions to run this test more quickly using concurrency.

13.1.2 Using concurrency

The application can leverage concurrency by updating the inventory in parallel. Figure 13.3 shows this principle by using the Concurrent Consumers EIP.

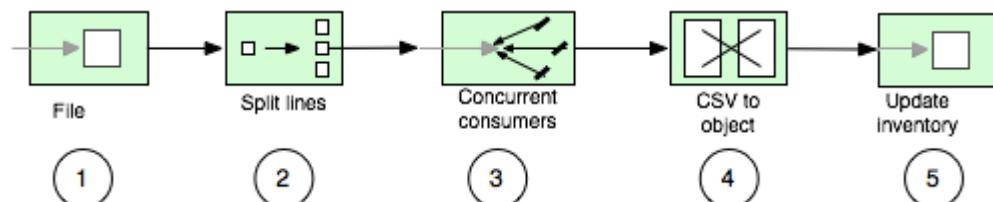


Figure 13.3 Using the Concurrent Consumers EIP to leverage concurrency and process inventory updates in parallel

As you can see in figure 13.3, the idea is to use concurrency ③ after the lines have been split ②. 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 only possible if the system allows a client to send messages concurrently to it. There can be situations where a system does not permit concurrency, or it may only allow up to a certain number of concurrent messages. Check the SLA (service level agreement) for the system you integrate with. Another reason to disallow concurrency would be if the messages have to be processed in the exact order they are split. Let's try out three different 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 very 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?n] INFO - Done processing big file
Took 11 seconds
```

What happens is that 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 IO-bound (reading files and remotely communicating with the ERP system involves a lot of IO 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 Camel Thread 4 - 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 upwards.

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 they run at their own pace. The reason why they appear in order here is because 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 will create a thread pool suitable for executing many small tasks. The pool will automatically grow and shrink on demand.

To use this pool with the Splitter EIP, you need to configure it as shown here:

```
.split(body()).tokenize("\n").streaming().executorService(threadPool)
    .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, as shown:

```
<split streaming="true" executorServiceRef="myPool">
  <tokenize token="\n"/>
  <bean beanType="camelinaction.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
[e://target/invene] INFO - Done processing big file
Took 2 seconds
```

You may wonder why it's now so fast. The reason is that the cached thread pool is designed to be very 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 will 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. This means that the thread pool receives new tasks at a higher 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
[ ntry?noop=true] 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 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

SEDA (staged event-driven architecture) 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 it 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.

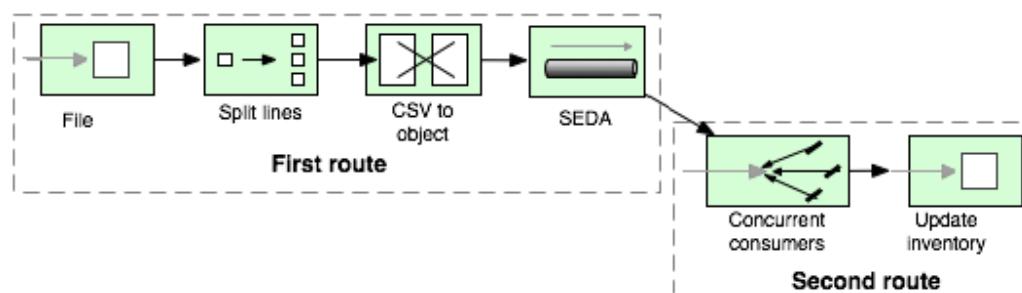


Figure 13.4 Messages pass from the first to the second route using SEDA. Concurrency is used in the second route.

Listing 13.2 shows how to implement this solution in Camel by 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("sed:update")
        .end()
        .log("Done processing file: ${header.CamelFileName}");
    from("sed:update?concurrentConsumers=20")
        .bean(InventoryService.class, "updateInventory");
}
```

①

① SEDA consumers using concurrency

By default, a seda consumer will only use one thread. To leverage concurrency, you use the concurrentConsumers option to increase the number of threads—to 20 in this listing ①.

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.

```
[read 20 - sed://update] INFO - Inventory 997 updated
[read 18 - sed://update] INFO - Inventory 998 updated
[read 9 - sed://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 leverage the concurrency features provided by the EIPs, such as the parallelProcessing on the Splitter EIP. It will leverage 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 different solutions for applying concurrency to an existing application, and they all greatly improve performance. We were able to reduce the 11-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 look at the default thread pool profile used by Camel and how to create custom thread pools using Java DSL and Spring XML.

13.2.1 Understanding thread pools in Java

A thread pool is a group of threads that are created to execute a number of tasks in a task queue. Figure 13.5 shows this principle.

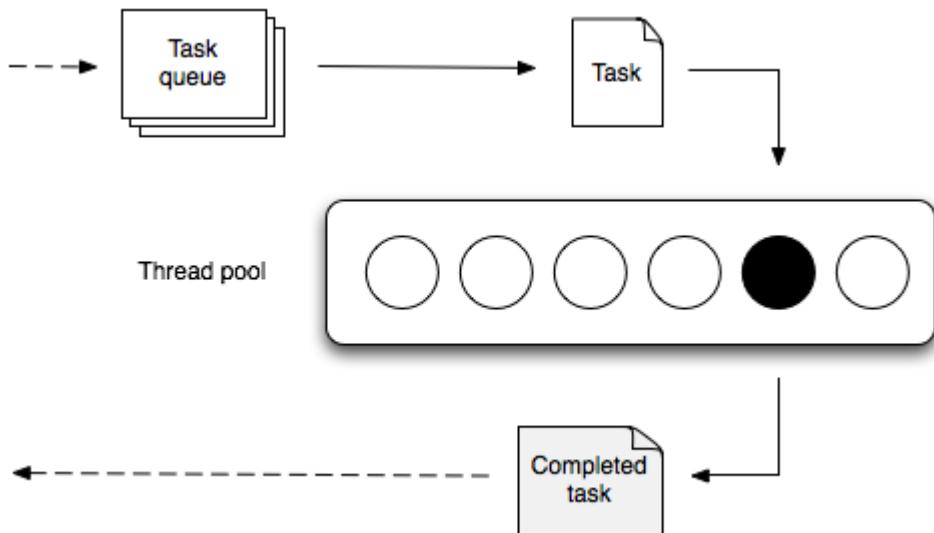


Figure 13.5 Tasks from the task queue wait to be executed by a thread from the thread pool.

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 leverages 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 depth about the Java concurrency API, we highly recommend the book *Java Concurrency in Practice* by Brian Goetz.

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>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. |
| <code>threadFactory</code> | <code>ThreadFactory</code> | Specifies a factory to use when a new thread is created. |

As you can see from table 13.1, there are many options you can use when creating thread pools in Java. To make it easier to create commonly used types of pools, Java provides `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, there are often additional tasks you must deal with. For example, it's important to ensure 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 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 name the threads using a common naming standard, 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 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 consumer.

```
[Thread 0] DEBUG - Total 3 files to consume
```

TIP Camel uses a customizable pattern for naming threads. The default pattern is "Camel \${camelId} thread \${counter} - \${name}". A custom pattern can be configured using ExecutorServiceStrategy.

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 Camel thread pool profiles

Thread pools aren't created and configured directly, but via the configuration of *thread pool profiles*. A thread pool profile is a profile that dictates how a thread pool should be created, based on a selection of the options listed earlier in table 13.1.

Thread pool profiles are organized in a simple two-layer hierarchy with custom and default profiles. There is 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 |
|----------------|---------------|---|
| poolSize | 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 | 60 | Idle threads are kept alive for 60 seconds, after which they're terminated. |
| maxQueueSize | 1000 | The task queue can contain up to 1000 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 the default values in table 13.3, 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 different values: `Abort`, `CallerRuns`, `DiscardOldest`, and `Discard`. The `CallerRuns` option will use the caller thread to execute the task itself. The other three options will either abort by throwing an exception, or discard an existing task from the task queue.

If the `maxQueueSize` option is configured to 0, then that means there is no task queue in use. So when a task is submitted to the thread pool, the task is only processed if there is an available thread in the pool. If there is no free thread then the rejection policy decides what happens. For example to use the caller thread, or fail with an exception etc. This is demonstrated later in section 13.3.1 where we process files concurrently without a task queue in use.

There is no one-size-fits-all solution 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 the `ThreadPoolProfile` starting from `CamelContext`. The following code shows how to change the maximum pool size to 50.

```
ExecutorServiceStrategy strategy = context.getExecutorServiceStrategy();
ThreadPoolProfile profile = strategy.getDefaultThreadPoolProfile();
profile.setMaxPoolSize(50);
```

The default `ThreadPoolProfile` is accessible from `ExecutorServiceStrategy`, which is an abstraction in Camel allowing you to plug in different thread pool providers. We'll cover `ExecutorServiceStrategy` in more detail in section 13.2.4.

In XML DSL, you configure the default thread pool profile using the `<threadPoolProfile>` tag:

```
<camelContext id="camel" 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 additional options if you want to override any of the other options from table 13.3.

There are situations where one profile isn't sufficient, so 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 `ThreadPoolProfileSupport` class:

```
ThreadPoolProfile custom = new ThreadPoolProfileSupport("bigPool");
custom.setMaxPoolSize(200);
context.getExecutorServiceStrategy().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 by 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="camelinaction.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 if there is 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 which 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 id="camel" 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 to leverage 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("MyPool");
```

The `ThreadPoolBuilder` requires `CamelContext` in its constructor, because it will 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="15" maxQueueSize="250"/>
    <route>
        <from uri="direct:start"/>
        <to uri="log:start"/>
        <threads executorServiceRef="myPool">           ①
            <to uri="log:hello"/>
        </threads>
    </route>
</camelContext>
```

① Using thread pool in the route

As you can see, the `<threadPool>` is used inside a `<camelContext>` tag. This is 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 we configured in the `<threadPool>` tag which in the example is `Cool` as shown in bold:

```
[Camel 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 are 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 `ExecutorServiceStrategy`, which defines a pluggable API for using thread pool providers.

13.2.4 Using ExecutorServiceStrategy

The `org.apache.camel.spi.ExecutorServiceStrategy` interface defines a pluggable API for thread pool providers. Camel will, by default, use the `DefaultExecutorServiceStrategy` 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 `ExecutorServiceStrategy` to work with the provider.

In this section, we'll show you how to configure Camel to use a custom `ExecutorServiceStrategy`, leaving the implementation of the provider up to you.

CONFIGURING CAMEL TO USE A CUSTOM EXECUTORSERVICESTRATEGY

In Java, you configure Camel to use a custom `ExecutorServiceStrategy` via the `setExecutorServiceStrategy` method on `CamelContext`:

```
CamelContext context = ...
context.setExecutorServiceStrategy(myExecutorServiceStrategy);
```

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.MyExecutorServiceStrategy"/>
```

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 EXECUTORSERVICESTRATEGY IN A CUSTOM COMPONENT

The `ExecutorServiceStrategy` 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 `ExecutorServiceStrategy`.

Listing 13.3 Using ExecutorServiceStrategy 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
    }

    protected void doStart() throws Exception {
        super.doStart();
        executor = getCamelContext().getExecutorServiceStrategy()
            .newScheduledThreadPool(this,
                "MyBackgroundTask", 1); ②
        executor.scheduleWithFixedDelay(this, 1, 1, TimeUnit.SECONDS);
    }

    protected void doStop() throws Exception {
        getCamelContext().getExecutorServiceStrategy().shutdown(executor);
        super.doStop();
    }
}
```

- ① Runs scheduled task
- ② Creates scheduled thread pool

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 `ExecutorServiceStrategy` ② and schedule it to run the task ① once every second using the `scheduleWithFixedDelay` method.

The source code for the book 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 Thread 0 - MyBackgroundTask] INFO  MyComponent - I run now
[Camel 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 leverage 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 take a 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. In this section, we'll take a look at them and the benefits they offer.

Table 13.4 EIPs in Camel that supports 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 to delay messages during routing. The delay can either be synchronous where the current thread is blocked, or asynchronous where the delay is using a scheduled thread pool to continue routing in a future time. It is only in the latter situation the Delayer EIP can process messages in parallel due 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 to throttle messages during routing, where some messages may be held back when the throttling limit has been reached. The throttling can either be synchronous where the current thread is blocked, or asynchronous where the throttling is using a scheduled thread pool to continue routing in a future time. It is only in the latter situation the Throttler EIP can process messages in parallel due the usage of the scheduled thread pool. This is similar to how the Delayer EIP works with parallel processing. |
| Wire Tap | 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. We 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 you used the Splitter EIP.

In the following three sections, we'll look at 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 fine-grained definition of the thread pool to be used. These additional options are listed 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="15"
            maxQueueSize="250">
            <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.

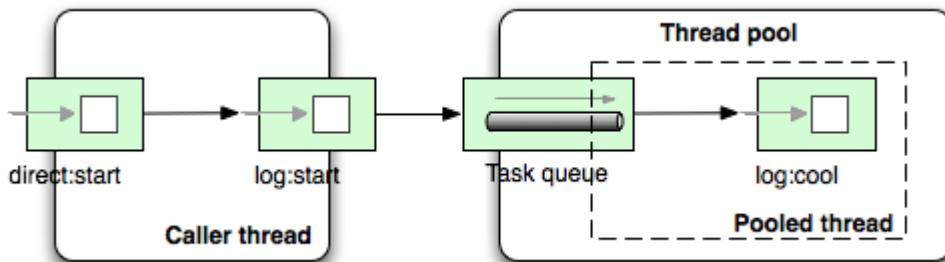


Figure 13.6 Caller and pooled threads are in use when a message is routed.

There will be two threads active when a message is being routed. The caller thread will hand over the message to the thread pool. The thread pool will then find 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 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.

You can use the Threads EIP to achieve concurrency when using Camel components that don't offer concurrency.

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. By using the Threads EIP, you can allow the picked up files to be processed concurrently.

Listing 13.4 shows an example how to do that with XML DSL.

Listing 13.4 Processing files concurrently with Thread EIP

```
<bean id="delayProcessor" class="camelaction.DelayProcessor"/>

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

    <route id="myRoute" autoStartup="false">
        <from uri="file:target/inbox?delete=true"/>
        <log message="About to process ${file:name} thread ##${threadName}" /> ①
        <threads poolSize="10" maxQueueSize="0">
            <log message="Start ${file:name} thread ##${threadName}" /> ②
            <process ref="delayProcessor"/>
            <log message="Done ${file:name} thread ##${threadName}" /> ③
        </threads>
        <to uri="log:done?groupSize=10"/> ④
    </route>
</camelContext>
```

- ① Consuming files from the starting directory
- ② Using Threads EIP to process the files in parallel
- ③ Random delay processing the files to simulate CPU processing
- ④ Log average processing speed per 10 files

This route starts from a file directory ① 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, we use threads ② configured with a thread pool of 10 active threads and no task queue. By setting `maxQueueSize` to 0 we do not use any in-memory task queue in the thread pool. That means the file consumer will only pickup new files, when there is a thread available at threads EIP to process the file. By default the thread pool would otherwise have a `maxQueueSize` of 1000 as outlined in table 13.3. A processor is used to delay processing each file, otherwise the

files are all processed to fast. A log endpoint ④ is used to log the average processing speed per 10 files.

The source code for this book contains this example in chapter13/pools which you can run 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 threads EIP, and as well pay attention to the logging output, to see which thread is doing what.

Let's look at how Rider Auto Parts improves performance by leveraging 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 different systems to gather an overview of the order. Figure 13.7 illustrates this.

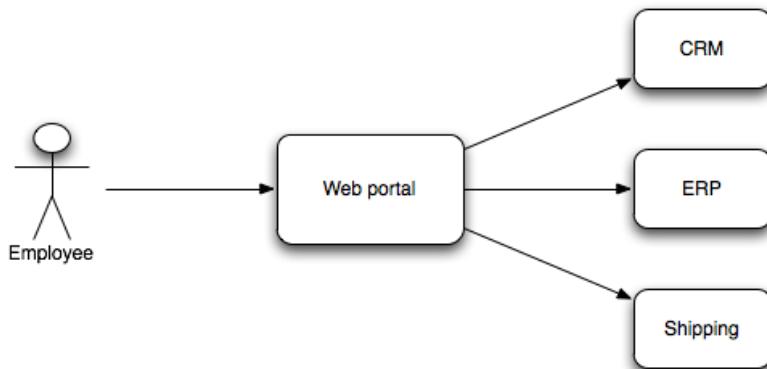


Figure 13.7 The web portal gathers information from three systems to compile the overview that's presented to the employee.

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 leveraging 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 order is always fixed in the sequential order defined by the Camel route.

The source code for the book 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 have 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 leverages 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 will 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>

```

```
</route>
</camelContext>
```

The source code for the book 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 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 is being tapped, and routes the 2nd message concurrently. If the message body is streaming based (e.g. `java.io.InputStream` type), then you should consider enabling stream caching that allows concurrent access to the stream. We will cover more about stream caching in chapter 15.

Wire tap is not only useable for tapping messages during routing. The pattern can also be used in the scenario where 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 acknowledgement 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.

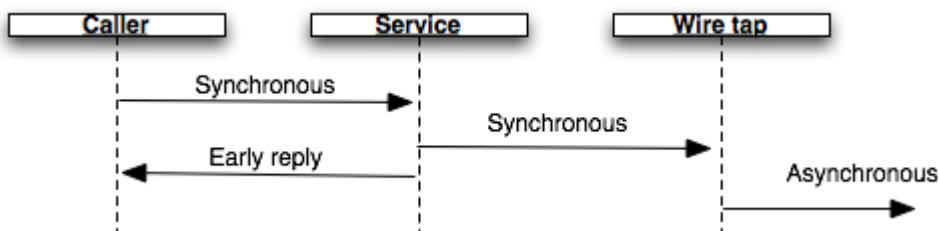


Figure 13.8 A synchronous caller invokes a Camel service. The service lets the wire tap continue processing the message asynchronously while the service returns an early reply to the waiting caller.

Implementing this example as a Camel route with the Java DSL can be done as shown in listing 13.5:

Listing 13.5 - Using WireTap to return an early reply message to the caller using Java DSL

```
from("jetty:http://localhost:8080/early").routeId("input")
    .wireTap("direct:incoming") ①
    .transform().constant("OK"); ②

from("direct:incoming").routeId("process") ③
    .convertBodyTo(String.class)
    .log("Incoming ${body}")
    .delay(3000)
    .log("Processing done for ${body}")
    .to("mock:result");
}
```

- ① Tap incoming message
- ② Return early reply
- ③ Route that processes the message asynchronously

You leverage the Wire Tap EIP ① to continue routing the incoming message in a separate thread, in the process route ③. This gives room for the consumer to immediately reply ② to the waiting caller.

Here's an equivalent example using XML DSL as written in listing 13.6:

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"/> ①
        <transform>
            <constant>OK</constant> ②
        </transform>
    </route>

    <route routeId="process">
        <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>
```

- ① Tap incoming message
- ② Return early reply
- ③ Route that processes the message asynchronously

The source code for the book 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 Thread 0 - WireTap] INFO - Incoming Hello Camel
11:18:15 [main] INFO - Caller finished calling Camel and received reply: OK
11:18:18 [Camel Thread 0 - WireTap] INFO - Processing done for Hello Camel
```

Notice in the console output how the caller immediately receives a reply within the same second it 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, which causes it to only be able to read the message once. Or, instead of converting the body, you could enable stream caching—we'll cover stream caching 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 all about how scalability works with Camel and how you can build custom components that would support scaling.

13.4 The asynchronous routing engine

Camel uses its routing engine to route messages either synchronously or asynchronously. In this section we focus on scalability and 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 you could increase throughput by leveraging concurrent processing. In that sense, the application was scalable, as 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 IO-bound processes, and because it could leverage thread pools to distribute work.

In this section, we'll look at scalability from a different angle. We'll look at 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.

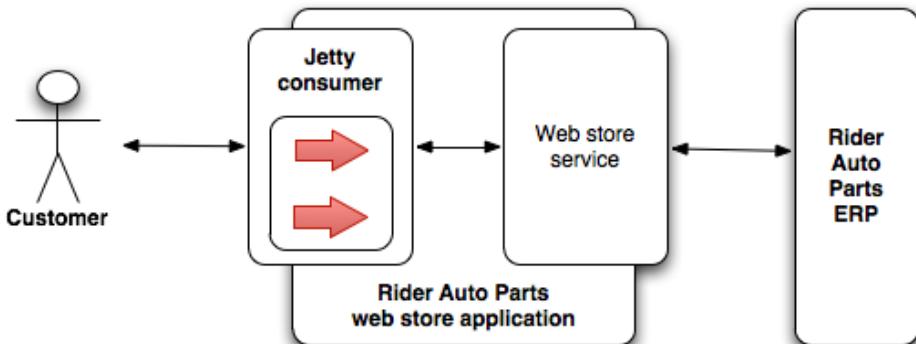


Figure 13.9 The Rider Auto Parts web store communicates with the ERP system to gather pricing information.

A 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 there’s one function that involves calculating pricing information for customers. The pricing model is complex and individual for 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 being calculated by the ERP system, the web store has to wait until the reply comes back, before it returns its response to the customer.

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 there have been problems 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 means each request that gathers pricing information has to wait (the thread is blocked) an average of 5 seconds for the reply to come back. 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.

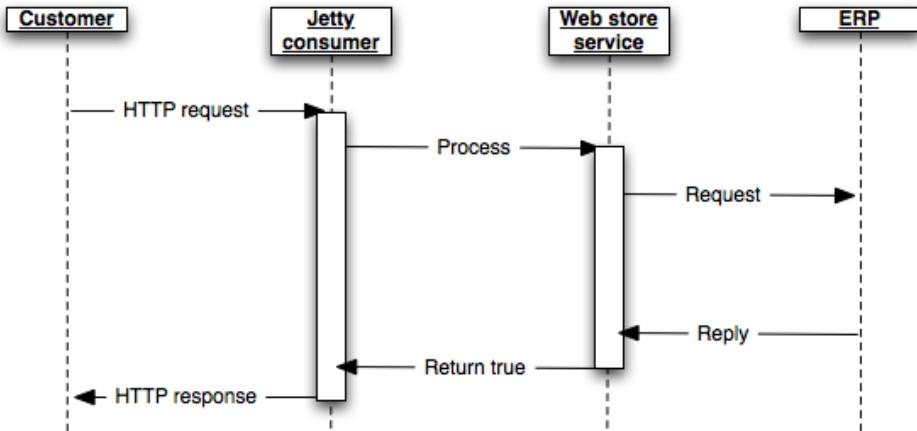


Figure 13.10 A scalability problem illustrated by the thread being blocked (represented as white boxes) while waiting for the ERP system to return a the reply.

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 look at 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 by using an asynchronous processing model. Figure 13.11 shows the principle.

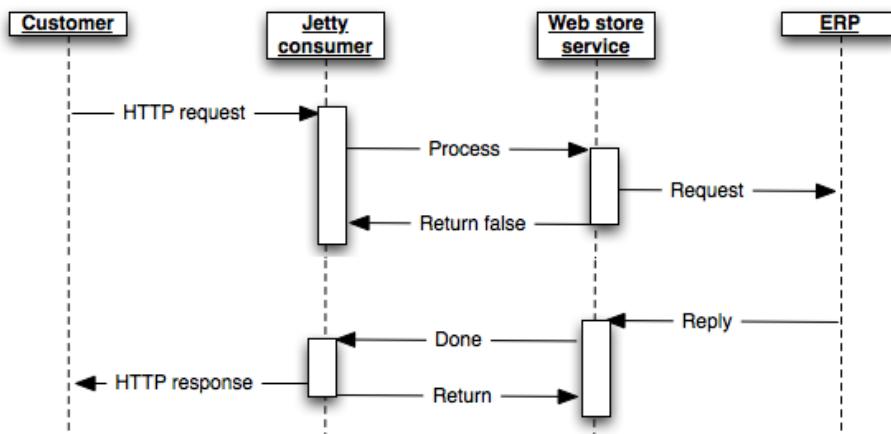


Figure 13.11 The scalability problem is greatly improved. Threads are much less blocked (represented by white boxes) when you leverage asynchronous communication between the systems.

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). In fact, there are no threads 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 utilization as much as it would otherwise do. The threads in the web store are much less IO-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.

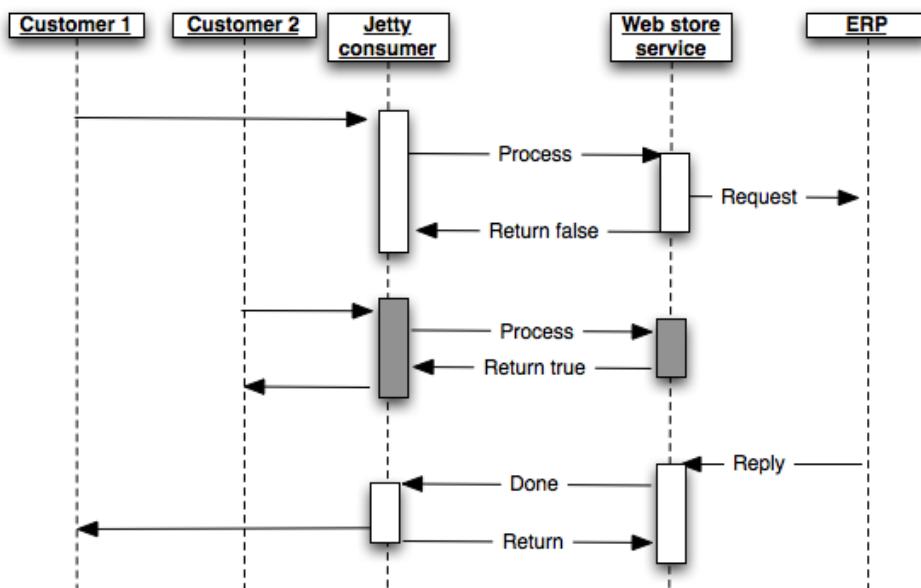


Figure 13.12 The same thread services multiple customers without blocking (white and grey boxes) and without impacting response times, resulting in much higher scalability.

In this situation, customer 1 sends a request that requires a price calculation, so the ERP system is invoked asynchronously. A short while thereafter, customer 2 sends a request that can be serviced directly by the web shop service, so it doesn't leverage 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.

In this example, you can successfully process two customers without any impact on their response time. You've achieved higher scalability.

In the next section, we'll look under the hood to see how this is possible in Camel 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.

There is a growing number of Camel components that support this, which is listed on the Camel website at: <http://camel.apache.org/asynchronous-routing-engine.html>

In order 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 13.16. 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 will 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. In the next section, we'll walk through this API.

13.4.4 Asynchronous API

Camel supports an asynchronous processing model, which we refer to as the asynchronous routing engine. There are advantages and disadvantages of using asynchronous processing, 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 |
|---|---|
| <ul style="list-style-type: none"> Processing messages asynchronously doesn't use up threads, forcing them to wait for processors to complete on blocking calls. It increases the scalability of the system by reducing the number of threads needed to manage the same workload. | <ul style="list-style-type: none"> Implementing asynchronous processing is more complex. |

The asynchronous processing model is manifested by an API that must be implemented to leverage asynchronous processing. You've already seen a glimpse of this API in figure 13.16; 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);
}
```

The `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 will 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's exception property.
- The `process` method must know whether it will 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.

In the next section, we'll cover an example of how to implement 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 source code for the book 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. Listing 13.7 shows how the `ErpProducer` is implemented.

Listing 13.7 ErpProducer using the asynchronous processing model

```

import java.util.concurrent.ExecutorService;
import org.apache.camel.AsyncCallback;
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultAsyncProducer;

public class ErpProducer extends DefaultAsyncProducer {          ①
    private ExecutorService executor;

    public ErpProducer(Endpoint endpoint) {
        super(endpoint);
    }

    protected void doStart() throws Exception {
        super.doStart();
        this.executor = getEndpoint().getCamelContext()           ②
            .getExecutorServiceManager().newFixedThreadPool(this, "ERP", 10);
    }

    protected void doStop() throws Exception {
        super.doStop();
        getEndpoint().getCamelContext()
            .getExecutorServiceManager().shutdown(executor);
    }

    public boolean process(final Exchange exchange,
                          final AsyncCallback callback) {           ③
        executor.submit(new ERPTask(exchange, callback));
        log.info("Returning false");
        return false;                                     ④
    }

    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);
            } catch (InterruptedException e) {
                // ignore
            }
            log.info("ERP reply received");
            String in = exchange.getIn().getBody(String.class);
            exchange.getOut().setBody(in + ";516");           ⑤
            log.info("Continue routing");
            callback.done(false);                            ⑥
        }
    }
}

```

- 1 Extends DefaultAsyncProducer
- 2 Thread pool used for simulate asynchronous tasks
- 3 Implements asynchronous process method
- 4 Returns false to use asynchronous processing
- 5 Sets reply on exchange
- 6 Notifies callback reply is ready

When implementing a custom asynchronous component, it's most often the `Producer` that leverages 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 ; this means you need to create a thread pool in the `doStart` method of the producer. To support the asynchronous processing model, the `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 .

According to the rules, when you're using `AsyncProcessor` the `callback` must be notified when you're done with a matching synchronous parameter ⑥ . 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 leverage 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:

```
2016-07-17 11:41:42 [      qtp1444378545-11] INFO  input
- Exchange[ExchangePattern:InOut, Body:1234;4;1719;bumper]
2016-07-17 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:

```
2016-07-17 11:49:48 [      qtp515060127-11] INFO  input
- Exchange[ExchangePattern:InOut, Body:1234;4;1719;bumper]
2016-07-17 11:49:48 [      qtp515060127-11] INFO  ErpProducer
- Returning false (processing will continue asynchronously)
2016-07-17 11:49:48 [Camel Thread 0 - ERP] INFO  ErpProducer
- Calling ERP
2016-07-17 11:49:53 [Camel Thread 0 - ERP] INFO  ErpProducer
- ERP reply received
2016-07-17 11:49:53 [Camel Thread 0 - ERP] INFO  ErpProducer
```

```
- Continue routing
2016-07-17 11:49:53 [Camel Thread 0 - ERP] INFO  output
- Exchange[ExchangePattern:InOut, Body:1234;4;1719;bumper;516]
```

This time there are two threads 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 the `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 highjack 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 is a price to pay, which is what we will discuss next.

13.4.6 Potential issues when using an asynchronous component

You may have seen from implementing a custom asynchronous component, such as shown in listing 13.7, that it's a bit more complicated to implement this correctly. In particular understanding the importance of making sure to call the `done` method on the `AsyncCallback` parameter ❶ . Calling the `done` method is what triggers the asynchronous routing engine in Camel to continue routing the exchange.

There are a number of potential issues causing the `done` method to never be called such as:

1. A bug in the code causing the `done` method to never be called.
2. The remote system never returns with a reply causing the task to not finish.
3. The `done` method is called but with wrong value (should use `false`)

We will dive into each of these items listed above in the following to give you some advice on how to reduce the risk of this happening when you implement a custom Camel component.

ENSURE THE DONE METHOD IS CALLED

You can help ensure that the `done` method is invoked by using `try .. finally` code style. For example in listing 13.7, the `ERPTask` class could be implemented with that as shown in listing 13.8.

Listing 13.8 - Using try .. finally to ensure done method is called

```
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() {
        try {
```

1

```

        log.info("Calling ERP");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            // ignore
        }
        log.info("ERP reply received");
        String in = exchange.getIn().getBody(String.class);
        exchange.getOut().setBody(in + ";516");      ③
        log.info("Continue routing");
    } finally {
        callback.done(false);                      ②
    }
}
}

```

- ① try block starts
- ② finally block to ensure done method is called
- ③ Imaginary if a NullPointerException was thrown here

The idea to use try .. catch is to ensure that the `done` method ② is always invoked even if the code throws an exception. For example suppose there was a bug in the code, causing a `NullPointerException` to be thrown at ③. Without the try .. catch block the `done` method would not be called, and the message would never continue being routed.

USE TIMEOUT WHEN CALLING A REMOTE SYSTEM

The custom component we implemented in listing 13.7 did not call a remote system. The implementation was intentionally kept simple to highlight the principle of how to structure a custom asynchronous Camel component. Now suppose the component did indeed call a remote system using some means of transport. Very often the means of transport using a third party library offers a way to specify a timeout value. When that is the case, then it's advised to configure the timeout value as an endpoint option on the Camel component.

Listing 13.9 shows the principle of using a timeout when calling a remote system.

Listing 13.9 - Use a timeout when calling a remote system

```

private class CallRemoteSystemTask implements Runnable {

    private final Exchange exchange;
    private final AsyncCallback callback;
    private final long timeout;

    private CallRemoteSystemTask(Exchange exchange, AsyncCallback callback,
                               long timeout) {
        this.exchange = exchange;
        this.callback = callback;
        this.timeout = timeout;
    }

    public void run() {
        try {
            String in = exchange.getIn().getBody(String.class);

```

```

Future<String> future = executor.submit(new ERPTask(in));      1
String response = future.get(timeout, TimeUnit.MILLISECONDS);  2

    if (response != null) {
        exchange.getOut().setBody(in + ";" + response);
    }
} catch (TimeoutException e) {                                     3
    exchange.setException(new ExchangeTimedOutException(exchange,
        timeout, "Timeout happened"));
} catch (Exception e) {                                         4
    exchange.setException(e);
} finally {
    callback.done(false);
}
}
}

```

- ➊ Simulate calling a remote system using asynchronous task
- ➋ Wait for the response within the timeout period
- ➌ In case of timeout set that as exception on the Exchange
- ➍ Ensure to set any other kind of exception on the Exchange
- ➎ Call done in the finally block

The example does not use a third party library for remote communication, but uses the concurrency framework from Java to simulate an asynchronous task ➊, which can issue a timeout error if the task does not respond within the timeout period ➋. If the task was not complete within the timeout period a `TimeoutException` was thrown at ➋ that we catch at ➌ and wrap that as a Camel specific timeout exception called `ExchangeTimedOutException`. We do this to ensure that timeout exceptions are uniform when using Camel as they are all using the same exception type. In case of any other kind of exception we catch those ➍ so the exception is stored on the `Exchange`, which allows Camel's error handler to react. And then we call the `done` method in the finally block ➎, that signals to Camel's routing engine to continue routing the message.

Java Concurrency Framework

The Java concurrency framework was introduced in Java 1.5 and refined over the following releases. It's designed as a set of building blocks for developers to build concurrent applications in a more safe way using higher level abstractions. The framework lives in the `java.util.concurrent` package and most known are the executor services (thread pools) which can run tasks concurrently, and the concurrent collections (list, set, map). It's outside the scope of this book to cover this framework which you can find information about on the internet. As far as books goes we highly recommend Brian Goetz book titled *Java Concurrency in Practice*.

The source code for the book contains an example of how to use a timeout in the chapter13/scalability directory. You can run the example using the following Maven goal:

```
mvn test -Dtest=TimeoutTest
```

CALLING DONE WITH THE RIGHT PARAMETER

The `done` method must be called with a boolean parameter, that matches what was returned from the process method. At first glance it may seem easy to just always return `false` because the component is asynchronous. But this can be a bit more tricky if the component does some work prior to the asynchronous task which could lead to an exception. Listing 13.10 illustrates how the producer of the component can either return `true` or `false`.

Listing 13.10 - BogusProducer idiom structure to handle exceptions prior to submitting asynchronous task submitted

```
public class BogusProducer extends DefaultAsyncProducer {

    public boolean process(final Exchange exchange,
                          final AsyncCallback callback) { ①

        try {
            String body = exchange.getIn().getBody(String.class);
            if (body.contains("Donkey")) {
                throw new IllegalArgumentException("No Donkeys allowed");
            }
        } catch (Exception e) {
            exchange.setException(e);
            callback.done(true); ③
            return true; ④
        }
    }

    BogusTask task = new BogusTask(exchange, callback); ⑤
    executor.submit(task);
    return false; ⑥
}
```

- ① The asynchronous process method of the producer
- ② Validate the input message body is not an illegal message
- ③ Catch validation exceptions and set on Exchange
- ④ Make sure to call callback done with true prior to returning
- ⑤ Return true to exit early
- ⑥ Everything is okay so submit the asynchronous task
- ⑦ And return false to process asynchronous

The source code for the book contains this example in the chapter13/scalability directory and you can try it from the command line using the following Maven goal:

```
mvn test -Dtest=DoneMethodTest
```

From listing 13.10 the custom component supports asynchronous processing by implementing the `AsyncProcessor` process method ①. In the producer we do some pre validation of the message body ②. In case of any validation error we throw an exception which we catch and set on the exchange ③. Because the producer has not done any asynchronous task, we can return early - this **must** be done by calling the `done` method with `true` prior to returning `true`

④ ⑤ . This is the opposite if there is no validation exception ⑥ , where we return false ⑦ prior to having the asynchronous task in the future call the `done` method with `false`.

In the following we will look at what happens if something goes wrong, for example a remote service never responds back causing the asynchronous component to never call the `done` method, which potentially could cause a thread to block forever.

13.4.7 Dangers with blocked threads

When you use various Camel components and EIP patterns then routing of the messages happens either using a synchronous, asynchronous or a combination of both threading model.

Chances for using a combination happens when you are using:

- Using request/reply messaging where Camel needs to return back a reply message when the routing of the message is complete.
- The routing is initially synchronous (or transaction is enabled)
- During routing of the message a asynchronous component is used
- The consumer thread would then need to block until the asynchronous component is done processing the message and signals this by calling the `done` method.

The items in the bulleted list are illustrated in the sequence diagram below in figure 13.13.

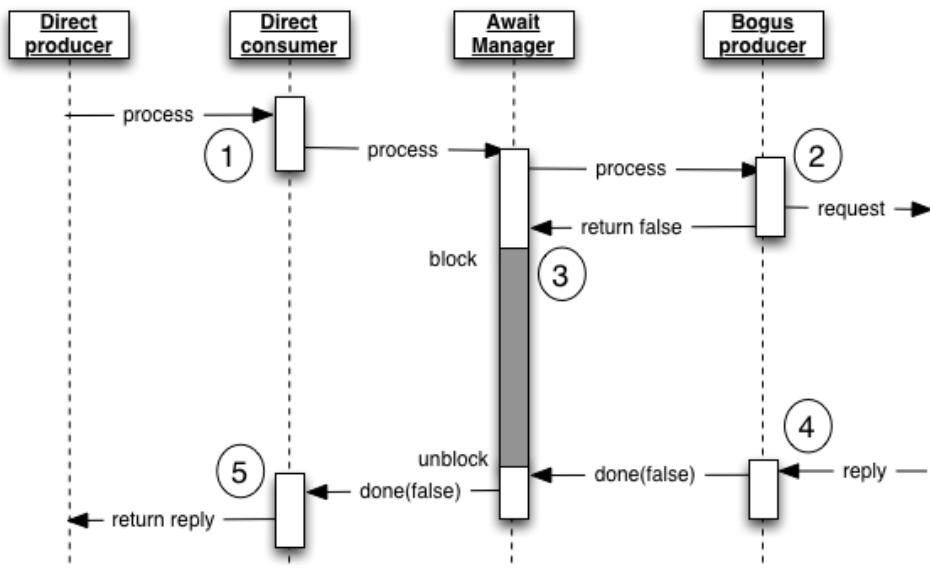


Figure 13.13 - During routing with a combination of synchronous and asynchronous components, then Camel potentially have to block thread(s) using the `AwaitManager`, which if poorly implemented could lead to threads blocked forever.

As the figure illustrates the route is starting from a synchronous component (such as the direct component) ① using request/reply (MEP is InOut). As the consumer of the route needs to send back a reply message when the routing is complete, it would potentially need to block until the routing is complete.

AwaitManager

During routing Camel keeps track of which thread is currently blocked using a manager with the cryptic name `DefaultAsyncProcessAwaitManager` - yes we acknowledge that naming in IT is a hard problem. This manager is available as JMX as well from Java API from a getter on `CamelContext`. The manager also offers information about each blocked thread such as the thread id and name, as well where the thread was blocked and for how long

During routing of the message an asynchronous component is used that uses non-blocking threading model ② . (For an example see ③ and ④ in listing 13.7). While the asynchronous component is processing the task in a non blocking model, the consumer would need to block while waiting for this task to complete ③ . When the asynchronous task is complete it signals this by calling the `done` method ④ , which will handover the `Exchange` and unblock the consumer thread ④ to let the thread continue routing which reaches the end of the route at the consumer, that sends back the reply message ⑤ .

The danger is that if the asynchronous task for some reason never signals that it is complete by calling the `done` method ④ , then the thread at ③ is blocked forever. The source code for this book contains an example in chapter13/scalability directory that demonstrates such a situation with a forever blocking (we simulate this by blocking for 5 minutes) thread. You can run this example from the command line using the following Maven goal:

```
mvn test -Dtest=BlockedTest
```

When the example is running, you can use a JMX console such as jconsole or hawtio to inspect the running Camel application, and be able to see how many threads are currently blocked, as shown in figure 13.14.

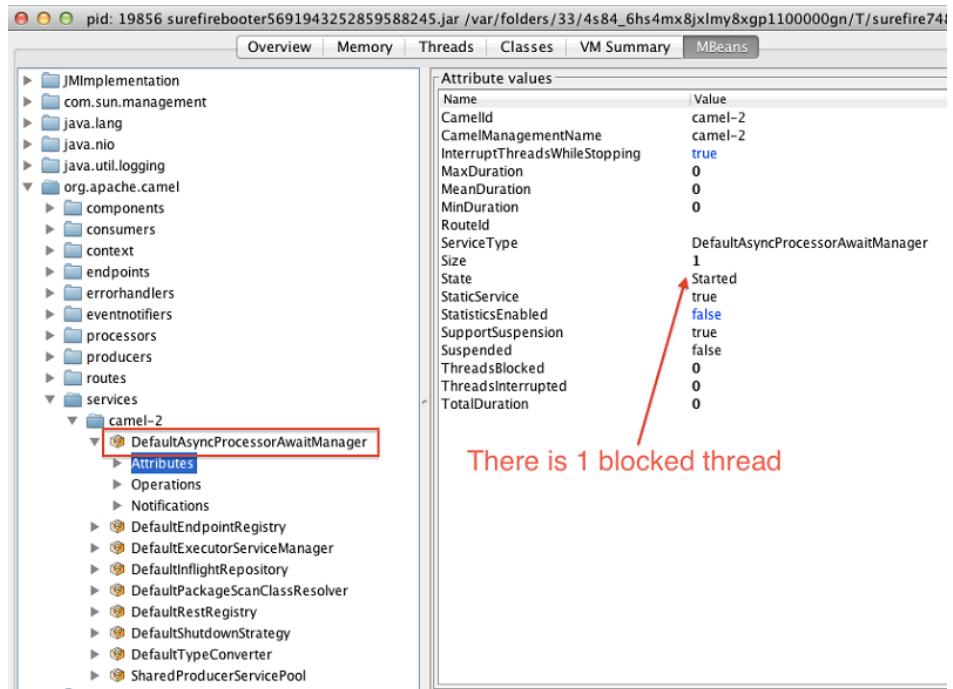


Figure 13.14 - Using jconsole to inspect the DefaultAsyncProcessAwaitManager MBean that provides details about blocked threads. There is currently 1 blocked thread indicated by the size attribute.

NOTE We will cover much more about managing Camel using JMX in chapter 16.

MANUALLY UNBLOCKING A BLOCKED THREAD

If a thread for some reason is never unblocked, then the manager can be used to unblock the thread by invoking the `interrupt` method with the exchange ID as parameter, as highlighted in the following figure 13.15.

You may want to do this if for example one or more threads are stuck on a production system, causing these threads to take up resources that are not released. Over time if more and more threads got stuck then this could lead to system instability as resources would become sparse as a JVM has an upper limit of the number of threads it can handle.

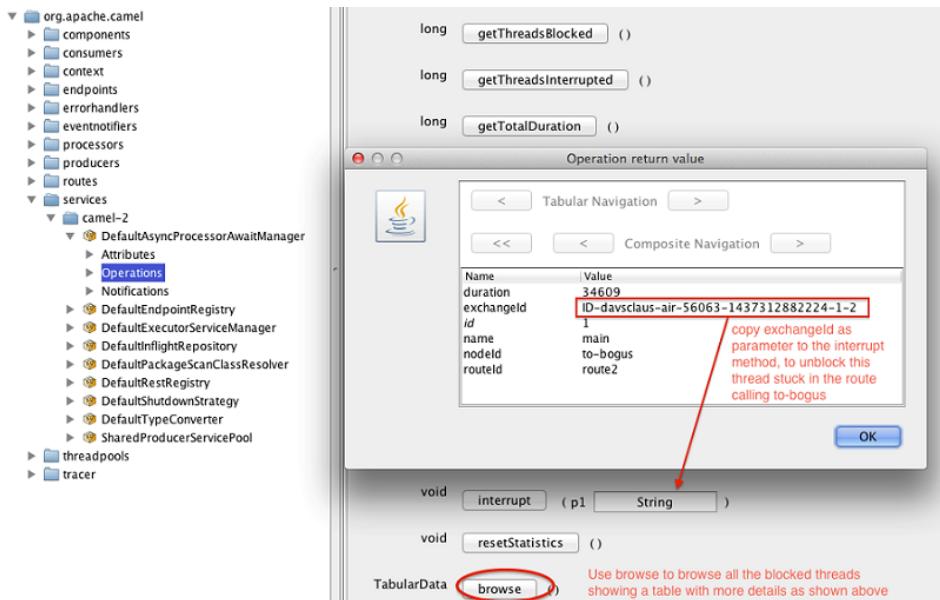


Figure 13.15 - Browsing all blocked threads using DefaultAsyncProcessorAwaitManager to show detailed information about the blocked thread. The thread can be unblocked by invoking the interrupt method with the exchange id as parameter.

When a thread is manually interrupted Camel will write to the log detailed information as shown in figure 13.16.

```
2015-07-19 15:36:13,509 [          RMI TCP Connection(2)-192.168.1.10]
WARN  aultAsyncProcessorAwaitManager - Interrupted while waiting
Blocked Thread
-----
Id:          1
Name:        main
RouteId:     route2
NodeId:      to-bogus
Duration:    85428 msec.

Message History
-----
RouteId      ProcessorId      Processor           Elapsed (ms)
[route2]     [route2]         [direct://start]   [ [ 85431]
[route2]     [log3]          [log]                [ [ 1]
[route2]     [to-bogus]       [bogus:foo]        [ [ 85430]

Exchange
-----
Exchange[ 
  Id          ID-davsclaus-air-56063-1437312882224-1-2
  ExchangePattern InOut
  Headers      {breadcrumbId=ID-davsclaus-air-56063-1437312882224-1-1}
  BodyType     String
  Body         ActiveMQ in Action
]
```

Figure 13.16 - When a thread is manually interrupted then detailed information is written to the log. We can see the thread was blocked when calling the processorId with "to-bogus" and was waiting for approx 85 seconds.

To unblock a thread is a manual procedure which you can do from JMX as shown in figure 13.15. An alternative is to use the hawtio web console, which has a unblock button as shown in figure 13.17. We will revisit hawtio in chapter 16 when we cover Camel management, and also in chapter 19 when we talk about Camel tooling.



The screenshot shows the hawtio web interface. The top navigation bar includes tabs for Camel, Connect, Dashboard, JMX, JUnit, Logs, and Threads. The Threads tab is selected. On the left, a sidebar tree view shows 'Camel Contexts' with 'camel-2' expanded, showing 'Routes' (with 'route2' selected), 'Endpoints', and 'MBEans'. The main content area displays a table titled 'Blocked' with the following data:

| Exchange Id | Route Id | Node Id | Duration (ms) | Thread Id | Thread name |
|--|----------|----------|---------------|-----------|-----------------|
| ID-davsclaus-air-59964-1438601606180-1-2 | route2 | to-bogus | 61590 | 24 | qtp555294517-24 |

A large 'Unblock' button is centered above the table. There are also 'Filter...' buttons at the top and bottom of the table.

Figure 13.17 - The hawtio web console showing a table with all blocked threads, which allows end users to select a thread and unblock easily by clicking the Unblock button.

You can also try this example using the hawtio web console by running the following Maven goal:

```
mvn hawtio:test
```

This will start the Camel application with hawtio embedded and hawtio should automatically open a web page and select the JUnit page. This page lists all the unit tests discovered in the JVM, which allows you to easily pick one or more tests and run the tests by clicking the *Run tests* button. While the tests are running, hawtio should show a Camel tab in the top menu bar which gives access to real time Camel information, such as the blocked threads page as shown in figure 13.17.

This concludes our coverage of scalability and the ups and downs with implementing custom asynchronous components with Camel and you have reached the end of this chapter.

13.5 Summary and best practices

In this chapter, we looked at thread pools, which are the foundation for concurrency in Java and Camel. We saw how concurrency greatly improves performance and we looked at all the possible ways to create, define, and use thread pools in Camel. You saw how easy it was to use concurrency with the numerous EIPs in Camel. You witnessed how Camel can scale up using asynchronous (non blocking) routing. This comes with a cost of complexity which we covered in great depth to learn about the pitfalls and best practices for implementing your custom Camel components. And finally we learned how you can detect if any threads are blocked (stuck) and how you can manually unblock these threads to let Camel continue routing the message.

Here are some best practices related to concurrency and scalability:

- *Leverage 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.* Only tweak the thread pools 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 are required to structure your component code according to a number of rules to ensure Camel can route the messages when data has been received or a timeout occurred. And as well to avoid any potential issue with threads getting stuck.

Now prepare for something completely different, as we are about to embark a journey into how to secure 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 be becoming 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 very 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 the company's finances.

With that said, it's important to note that Camel is by default **not** secured! There is a good reason for this: there are many angles to application security 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. However, authentication and authorization may be needed. Camel can help you implement as much or little security as you require, with relative ease. I say relative because security configuration can become quite complex just by nature.

In this chapter we'll be covering the main areas of security in Camel. We'll start off by covering how to secure sensitive information in your configuration. Next we'll cover security in

web services. Then we'll look at 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 you can secure your configuration.

14.1 Securing your configuration

As we saw back in chapter 9, externalizing your configuration and referencing with property placeholders is a great way 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 is a chance someone may view this information, having everything in plain text would give that person access to the remote resource. To prevent this 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 how we can use 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. We'll cover the runtime scenario (i.e. decrypting configuration) in the next section. 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 like:

```
[janstey@bender]$ cd apache-camel-2.15.2/
[janstey@bender]$ java -jar lib/camel-jasypt-2.15.2.jar -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 a password "secret" you would use the following options:

```
[janstey@bender]$ java -jar lib/camel-jasypt-2.15.2.jar -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 we'll need to use later when decrypting configuration. The output from the utility is the encrypted value of the "secret" password. We can just as easily go back to the original value like:

```
[janstey@bender]$ java -jar lib/camel-jasypt-2.15.2.jar -c decrypt -p supersecret -i
  "q+XT/4rR94ghCbNp5coaxg=="
Decrypted text: secret
```

In both cases, we did not 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 ⁴ [4](#). If you decide to use some other algorithm here, make sure to use the same when you are decrypting configuration within your route. Also, not all security providers provide the same set of algorithms so before configuring something different ensure all systems in your deployment can actually support the algorithm.

Now that we have our password encrypted, let's look at how we can use that within our Camel route.

14.1.2 Decrypting configuration

To use the camel-jasypt component within your Camel project you will of course need to add a dependency to it in your Maven POM like:

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-jasypt</artifactId>
<version>2.16.0</version>
</dependency>
```

You then need to configure the Camel Properties component to use the `PropertyParser` provided by `camel-jasypt`. In Spring XML, 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 id="camel" xmlns="http://camel.apache.org/schema/spring">
<propertyPlaceholder id="properties"
location="classpath:rider-test.properties"
propertiesParserRef="jasypt"/>
...
</camelContext>
```

In the `rider-test.properties` file, you can then define the encrypted properties using the special `ENC()` notation:

```
ftp.password=ENC(q+XT/4rR94ghCbNp5coaxg==)
```

⁴ <http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html>

The text inside “ENC(“ and ”)” is the “secret” password that we encrypted in the previous section. We can now use the encrypted property directly in the endpoint URI, as shown in bold in this route:

```
<camelContext id="camel" 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>
```

Of course, you can do all of this from the Java DSL as well. Listing 14.1 shows how this can be done.

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
        jasypt.setPassword("supersecret"); ②
        PropertiesComponent prop =
            context.getComponent("properties", PropertiesComponent.class);
        prop.setLocation("classpath:rider-test.properties");
        prop.setPropertiesParser(jasypt); ③
        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/target/outbox"); ④
            }
        };
    }
}
```

- ① Create the jasypt properties parser
- ② Set the encryption password
- ③ Use the jasypt properties parser so we can decrypt values
- ④ Reference the encrypted property with an endpoint URI

After creating the `CamelContext`, you create the `jasypt PropertyParser` ❶ and set it on the `PropertiesComponent` ❷. Of course, we also need to use the encryption password ❸ as we did in the Spring XML example before.

You can run try this example using the following Maven goals from the `chapter14/configuration` directory:

```
mvn test -Dtest=SecuringConfigTest
mvn test -Dtest=SpringSecuringConfigTest
```

Since you have your security hat on right now, you may have been thinking this plaintext encryption password is a security hole. You would be right of course! After all what would be the point of encrypting your remote service password when you then leave the encryption password in plain view?

Externalizing the encryption password

So how do you deal with a plaintext encryption password? The solution is to not define it in your application but 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 ❷, 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 would set an additional environment variable like:

```
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 we've secured our configuration, let's look at how to secure web services.

14.2 Web service security

You would be hard pressed to find any modern enterprise project that doesn't use web services of some sort. They're an extremely useful integration technology for distributed applications. Web services are often associated with service-oriented architecture (SOA), where 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 what operations you can call on a web service and what the input and output types are, among other things. Messages are typically XML, formatted to comply with the Simple Object Access Protocol (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, one of which is Web Services Security (WS-Security). The WS-Security support in CXF allows you to:

- Use authentication tokens
- Encrypt messages
- Sign messages
- Timestamp messages

To show these concepts in action, let's going back to Rider Auto Parts, where they need a new piece of functionality implemented. In chapter 2 you saw how 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 backend order processing systems needed to be a web service. Of course, we wouldn't want anonymous users to be able to submit orders so we're going to use WS-Security to help us out.

14.2.1 Authentication in web services

Clients of the new Rider Auto Parts web service have a really simple way to submit orders. The service endpoint in question looks like:

```
import javax.jws.WebService;
@WebService
public interface OrderEndpoint {
    OrderResult order(Order order);
}
```

They can simply call the order method with an Order object as the only argument. The Order object contains a part name, amount to order, and also the customer name. An OrderResult is returned and can tell the user whether the order succeeded or failed. Of course the big missing piece here is that it is 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 need to add authentication to this service.

Authentication is where a user proves who they are to a system using typically 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 like whether a username exists, a password is valid, or if the timestamp on the token is still valid. Looking at an 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" ② ase64Binary">Zi8UQpTp28/OeSNqIO9eTQ==</wsse:Nonce>
<wsu:Created>2015-10-25T18:06:46.052Z</wsu:Created>
</wsse:UsernameToken>
```

This is 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 off, let's take a look at how the server-side configuration will change when security is added. The route used to implement the web service won't actually change at all:

```

<route>
    <from uri="cxfr:bean:orderEndpoint" />
    <to uri="seda:incomingOrders" />
    <transform>
        <method beanType="camelinaction.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. We need to look at the CXF configuration to see how this authentication was set up. Listing 14.2 shows this.

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
    <constructor-arg>
        <map>
            <entry key="action" value="UsernameToken Timestamp"/> ②
            <entry key="passwordCallbackClass"
                value="camelinaction.wssecurity.ServerPasswordCallback"/> ③
        </map>
    </constructor-arg>
</bean>

<cxf:cxfEndpoint id="orderEndpoint"
    address="http://localhost:9000/order"
    serviceClass="camelinaction.order.OrderEndpoint">

    <cxf:inInterceptors>
        <ref bean="loggingInInterceptor"/>
    </cxf:inInterceptors>
</cxf:cxfEndpoint>
```

```

<ref bean="wss4jInInterceptor"/> ④
</cxf:inInterceptors>

<cxf:outInterceptors>
  <ref bean="loggingOutInterceptor"/>
</cxf:outInterceptors>
</cxf:cxfEndpoint>

```

- ① Create a WSS4JInInterceptor to process the WS-Security SOAP header
- ② Set the actions to use when processing the UsernameToken
- ③ Specify the Callback to use when checking the username and password
- ④ Add the WSS4JInInterceptor to the cxfEndpoint of the OrderEndpoint

Our 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 ① . 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 ② specifies what tasks will be executed using the interceptor. In our case, this will be to process a UsernameToken and also a timestamp. Take note of these actions as the client code will also need to provide them. The "passwordCallbackClass" ③ entry specifies a class that will be used to check the username and provide a password. In our case we'll simply be using a hard-coded username/password combination to demonstrate the flow. Such a callback handler class is shown in Listing 14.3.

Listing 14.3 A simple WS-Security callback for a server-side app

```

package camelinaaction.wssecurity;

import java.io.IOException;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler; ①
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.wss4j.common.ext.WSPasswordCallback;

public class ServerPasswordCallback implements CallbackHandler {
public ServerPasswordCallback() {
}

public void handle(Callback[] callbacks)
throws IOException, UnsupportedCallbackException {
WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];
if ("jon".equals(pc.getIdentifier())) { ②
pc.setPassword("secret"); ③
} else {
throw new IOException("Username incorrect!");
}
}
}

```

- ① We need to implement a CallbackHandler
- ② Check the username
- ③ Set the password on the callback if the username is valid

First off our ServerPasswordCallback is implementing CallbackHandler ① , which has a single method "handle". In the handle method we need to check whether the username is valid ② and if so set the password on the callback ③ . The actual verification of the password is done by WSS4J under the covers. The password itself may not be stored in plain-text so actually comparing it may be an effort in itself - it's best to leave the checking up to WSS4J.

Now, Rider Auto Parts order web service has full authentication support, albeit for a single user "jon", but still full authentication support. Lets next see how clients of this webservice can add user credentials.

CLIENT-SIDE WS-SECURITY CONFIGURATION

Like the server-side case, the main thing we are doing to add WS-Security authentication to our application is by adding a WSS4J interceptor to our CXF based webservice. It's easiest to explain with code of course so let's look at Listing 14.4 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<Interceptor<?
extends Message>>();

    // Define WSS4j properties for flow outgoing
    Map<String, Object> outProps = new HashMap<String, Object>();
    outProps.put("action", "UsernameToken Timestamp"); ①
    outProps.put("user", user); ②
    outProps.put("passwordCallbackClass", passwordCallbackClass); ③

    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
    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
    assertEquals("OK", reply.getMessage());
}
```

① Set the actions to use when processing the UsernameToken

② Specify the user

- ③ Specify the Callback to use to grab the password
- ④ Use a CXF factory bean to create the OrderEndpoint
- ⑤ Place an order using the OrderEndpoint

So before we create a business-as-usual JAX-WS web service using a CXF factory bean ④ we need to configure a WSS4JOutInterceptor. We use the same actions as on the server-side ① . That is, we want to specify a UsernameToken and also a Timestamp so a passed authentication doesn't last indefinitely. We are on the client side now so we need to specify the user we want accessing the remote service. We do this by adding a "user" entry for the user named "jon" ② . We specify a different callback handler on the client side ③ which we'll discuss next. Finally, we place an order using the OrderEndpoint created by the CXF JAX-WS factory bean ④ – if not obvious, this will invoke the remote web service and return a result. All the marshalling and networking is handled by CXF under the hood. [TODO web services chapter ref](#)

The callback handler for the client-side is a little different in that we are simply providing a password to use when calling the web service. Listing 14.5 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 ClientPasswordCallback() {
    }

    public void handle(Callback[] callbacks) throws IOException,
    UnsupportedCallbackException {
        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];
        pc.setPassword("secret"); ①
    }
}
```

- ① Specify the password to use when invoking the webservice

This overly simplified callback handler gets the job done for our hardcoded example but of course you wouldn't want to always return the same password for every user.

You can try this example out for yourself by navigating to the chapter14/webservice directory and executing the following command:

```
mvn test -Dtest=WssAuthTest
```

In particular, note how the web service invocation will fail with a `javax.xml.ws.soap.SOAPFaultException` when either the username or password is incorrect. Recall this is how we designed the server-side callback handler in Listing 14.3. Of course, with hardcoded usernames and passwords this is highly unlikely. Let's look at how we can integrate our webservice with container authentication.

14.2.2 Authenticating web services using JAAS

The developers at Rider Auto Parts have successfully demonstrated their authenticated web service to management. So now they are tasked with authenticating against the users available in their company-wide container standard: Apache Karaf. Users in Karaf can come from various sources like LDAP, database, properties file, etc. All of these are accessed via the Java Authentication and Authorization Service (JAAS).

This process is made easy by using an existing class from WSS4J. Recall how in Listing 14.3 the server-side `CallbackHandler` was simply setting the password text on the `WSPasswordCallback`, without checking for its validity? We mentioned how we were 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. Listing 14.6 show you how.

Listing 14.6 Using a JAASUsernameTokenValidator to authenticate against JAAS

```
<bean id="karafJaasValidator" ①
  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>

<cxf:cxfEndpoint id="orderEndpoint" address="/order" ④
  serviceClass="camelinaction.order.OrderEndpoint">

  <cxf:inInterceptors>
    <ref component-id="loggingInInterceptor" />
    <ref component-id="wss4jInInterceptor" />
  </cxf:inInterceptors>
  <cxf:outInterceptors>
    <ref component-id="loggingOutInterceptor" />
  </cxf:outInterceptors>

  <cxf:properties>
    <entry key="ws-security.ut.validator"> ③
      <ref component-id="karafJaasValidator" /> ③
    </entry>
  </cxf:properties>

```

```
</entry> ③
</cxf:properties>

</cxf:cxfEndpoint>
```

- ① Point the JAASUsernameTokenValidator to the Karaf container JAAS context
- ② The WSS4JInInterceptor should check for UserTokens
- ③ Override the default password validator with JAASUsernameTokenValidator
- ④ Use relative path for web service

First off we construct a org.apache.wss4j.dom.validate.JAASUsernameTokenValidator that is configured to check UsernameToken credentials against a javax.security.auth.login.LoginContext with name “karaf” ① . When deployed into Apache Karaf, this will pick up whatever login module it is using for authentication (LDAP, database, properties file, etc). Now, when we create the WSS4JInInterceptor ② , it is different from before. The main difference is that we are now supplying no CallbackHandler. This is actually a feature of Apache CXF – since we provided no CallbackHandler and our UsernameToken has a password set, CXF creates a simple CallbackHandler for us. This simple CallbackHandler just sets the password on the WSCallbackHandler, similar to what we were doing manually before. Finally, we need to override the default password validator with our JAAS-powered one. We can do this by using the “ws-security.ut.validator” CXF configuration property ③ . With this new configuration our web service is now authenticated against container credentials.

NOTE Web services deployed into a container like Apache Karaf typically reuse an HTTP service that is already running. In Karaf’s case, HTTP services are by default on port 8181 and CXF web services are registered under the “/cxf” path. So for our web service in Listing 14.6, the “/order” path ④ will be available at <http://localhost:8181/cxf/order>.

To demonstrate this we actually need to deploy the example into Apache Karaf. At the time of writing, we used the latest version which was Apache Karaf 4.0.1. So you first start Apache Karaf using the following command:

```
bin/karaf
```

Then install Camel and CXF WS-Security support in Karaf:

```
feature:repo-add camel 2.16.0
feature:install camel-cxf cxf-ws-security camel-blueprint
```

... where 2.16.0 is the Camel version to install.

Now we need to install our Rider Auto Parts web service in Karaf, which can be done using the chapter14/webservice-karaf example from the source code of the book. At first we 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:

```
install -s mvn:com.camelinaction/chapter14-webservice/2.0.0
install -s mvn:com.camelinaction/chapter14-webservice-karaf/2.0.0
```

.. which will install and start the example (the `-s` flag refers to start).

The chapter14/webservice-karaf directory also contains a client which we can use to access the web service deployed in Karaf. This client is shown in Listing 14.7.

Listing 14.7 A client to access the Rider Auto web service

```
public class Client {

    public static void main(String[] args) {
        List<Interceptor<? extends Message>> outInterceptors
            = new ArrayList<Interceptor<? extends Message>>();

        // Define WSS4j properties for flow outgoing
        Map<String, Object> outProps = new HashMap<String, Object>();
        outProps.put("action", "UsernameToken");
        outProps.put("user", "karaf"); ①
        outProps.put("passwordType", "PasswordText"); ②
        outProps.put("passwordCallbackClass",
                    "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());
    }
}
```

- ① Use the default Karaf user
- ② Use text based password to be compatible with the JAAS validator
- ③ Use a CallbackHandler that accepts a password from the console

Our simple Client class looks very similar to the web services test from the previous section. The first main difference is that we are specifying the default Karaf admin user "karaf" ① as the user. We then set the password type as text so that the WSS4J JAAS validator can compare it correctly ② . Finally we specify a CallbackHandler that reads the password from the console. This simple CallbackHandler is shown in Listing 14.8 below.

Listing 14.8 A CallbackHandler that reads passwords from the console

```
public class StdInPasswordCallback implements CallbackHandler {
    public StdInPasswordCallback() {
    }

    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 out 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 will need to enter "karaf" as the password. If you don't enter the correct password, a javax.xml.ws.soap.SOAPFaultException will be thrown from the CXF client.

The WS-Security support in CXF also allows you to do things like sign or encrypt your messages, among other things. For more information on how to configure these, please refer to the Apache CXF website: <http://cxf.apache.org/docs/ws-security.html>.

Now that we've secured our Rider Auto Parts web service, let's take a look at securing more general payloads we could be sending with Camel.

14.3 Payload security

There are two main options to look at when thinking about securing the message payloads you are sending with Camel. One involves encrypting the entire contents of the message; a very useful option when you are sending sensitive information that you would not like viewed by any listening parties. Now, 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 will encounter out in the wild. To see more about configuring TLS skip to section 14.3. There is a difference to handling encryption within Camel or delegating to the transport layer. Figures 14.1 and 14.2 illustrate this difference.

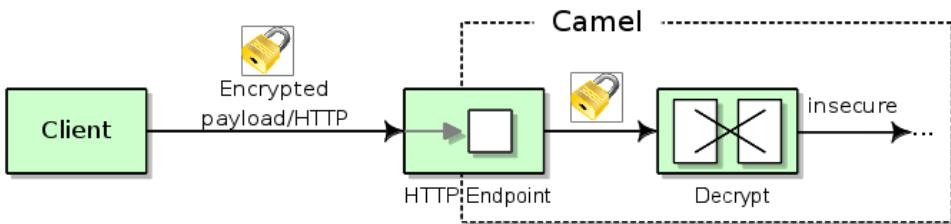


Figure 14.1 Using the camel-crypto data format you can handle encryption and decryption of payloads within your Camel route. This gives you full control over what transport you send over – like, an unsecured HTTP pipe.

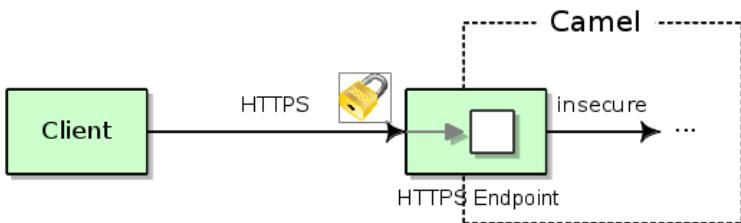


Figure 14.2 Using transport based security to handle payload encryption gives you less choice over what transport you can use because the underlying transport has to support JSSE. For a complete discussion on transport security please see section 14.3.

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 also 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 you sent the message and also that no one tampered with the message in transit. This is especially crucial in messages like financial transactions, where a change of message origin or modification of a decimal place say would have large consequences. Digital signatures are generated by using asymmetric cryptography, meaning the sender generates the signature using a private key that only he has. The receiver then uses a different publicly available key to verify that the message is indeed the same. The mathematics behind this relationship are quite complex, but fortunately you don't have to delve into that to make use of the algorithms.

For signing messages in Camel, there is the camel-crypto component. To use the camel-crypto component within your Camel project you will of course need to add a dependency to it in your Maven POM like:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-crypto</artifactId>
  <version>2.16.0</version>
</dependency>
```

Now before we can actually dig into what the camel-crypto component gives you, we're first going to 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. Just take note that if you do decide 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 – that is, 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 we are creating public/private key pair with alias "jon", expiration in 7300 days, and key password of "secret". The dname argument specifies the distinguished name of the certificate issuer. Finally, we specified cia_keystore.jks as the keystore file with a password of "supersecret". If we use the -list keytool command, we can see there is one key-pair in our keystore:

```
[janstey@bender]$ keytool -list -keystore cia_keystore.jks -storepass supersecret

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

jon, Jun 29, 2015, 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 us to start signing messages however, we need to extract the public key into a separate keystore since we can't give out our private key. This keystore that contains only public keys is called the truststore. To do this we 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 we can use the keys stored in the keystores in Camel, we 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:

```
@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 this can be done 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" />
```

Now that we have the key and trust stores ready we 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 would use `crypto:sign` to add a digital signature to the message. Using our previously generated keystore this would look something like:

```
from("direct:sign")
.to("crypto:sign://keystore?keyStoreParameters=#keystore&alias=jon&password=secret")
...
```

In Spring or blueprint XML this would look like:

```
<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 by using the default SHA1WithDSA algorithm and the private key with alias "jon" in the keystore loaded via the `KeyStoreParameters` in the `#keystore` reference.

Verifying this message later on is again a simple step in your route:

```

from("direct:verify")
.to("crypto:verify://keystore?keyStoreParameters=#truststore&alias=jon&password=secret")
...

```

In Spring or blueprint XML this would look like:

```

<route>
  <from uri="direct:verify"/>
  <to
uri="crypto:verify://keystore?keyStoreParameters=#truststore&alias=jon&password=secret"/>
  ...
</route>

```

The only difference from before is that now we are using the "verify" endpoint and we are using the truststore instead of the keystore. On success, nothing will happen and your route will continue on 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 and simulate a man-in-the-middle type attack by modifying the message body before it gets to the verify step. We'll use three routes to simulate this:

```

from("direct:sign")
.to("crypto:sign://keystore?keyStoreParameters=#keystore&alias=jon&password=secret")
.to("mock:signed")
.to("direct:mitm");
from("direct:mitm")
.setBody().simple("I'm hacked!")
.to("direct:verify");
from("direct:verify")
.to("crypto:verify://keystore?keyStoreParameters=#truststore&alias=jon&password=secret")
.to("mock:verified");

```

Now when sending a message to the "direct:sign" endpoint:

```

try {
  template.sendBody("direct:sign", "Hello World");
} catch (CamelExecutionException e) {
  assertEquals(SignatureException.class, e.getCause());
}

```

```
}
```

We can expect its body to be modified to "I'm hacked!" and then of course the verify step will fail. Notice how Camel threw a `CamelExecutionException` with the cause being a `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 we've stopped a man-in-the-middle attack using digital signatures, let's see how we can encrypt entire payloads.

14.3.2 Payload encryption

Sometimes just viewing the contents of a message can be harmful. 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 that we looked at previously, where the sender and receiver had different but complimentary keys (i.e. they were asymmetric). Since we just looked at an asymmetric example, let's take a look at how a symmetric case would work.

NOTE Asymmetric encryption is handled with the `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. To use the `camel-crypto` component within your Camel project you will of course need to add a dependency to it in your Maven POM like:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-crypto</artifactId>
  <version>2.16.0</version>
</dependency>
```

Also, like when signing messages with Camel we 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 we can't reuse those keys.

Generating and loading secret keys

To generate a secret key using `keytool` you can use the `-genseckeyp` command:

```
keytool -genseckeyp
  -alias ciassecrets
  -keypass secret
```

```
-keystore cia_secrets.jceks
-storepass supersecret
-storetype JCEKS
```

Here we are creating a secret key with alias "ciasecrets", and key password of "secret". Note that we can't use the default store type anymore – you need to use "JCEKS" for storing secret keys. Finally, we specified `cia_secrets.jceks` as the keystore file with a password of "supersecret".

As for loading the keystore in Camel, we need to load them in the same way as we did for message signing except that 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.setKeyType("JCEKS");
    KeyStore store = keystore.createKeyStore();
    secretKey = store.getKey("ciasecrets", "secret".toCharArray());
    registry.bind("secretKey", secretKey);
    return registry;
}
```

Here we use `KeyStoreParameters` again to do the grunt work of loading the keystore but we are adding the secret key to the registry instead of the `KeyStoreParameters`.

ENCRYPTING AND DECRYPTING PAYLOADS USING CAMEL-CRYPTO

Payload encryption in camel-crypto is implemented using data formats. You saw data formats used in depth in chapter 3. Basically, a call to `marshal` 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 we specify an encryption algorithm of Digital Encryption Standard (DES)⁵ and the secret key we created and loaded previously. Now you can use the data format in a route like:

```
from("direct:start")
    .marshal(crypto)
    .to("mock:encrypted")
    .unmarshal(crypto)
    .to("mock:unencrypted");
```

⁵ [2 All possible cipher algorithms are listed in the Java™ Cryptography Architecture Standard Algorithm Name Documentation - https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#ipher](https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#ipher)

In Spring or blueprint XML this would look like:

```
<bean id="secretKey" class="camelinaction.SpringMessageEncryptionTest" factory-
    method="loadKey" />

<camelContext id="camel" 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>
```

If it's not obvious, 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. Once you have your key loaded and data format set up, 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 your data is encrypted outside your Camel route is to make use of transport security, which is all about setting up the Transport Layer Security (TLS) protocol on the underlying transport.

14.4 Transport security

Many components in Camel allow you to configure Transport Layer Security (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. However not all 3rd party libraries that Camel integrate with make use of 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, a number of Camel components make use of this utility:

- camel-ahc
- camel-apns
- camel-box
- camel-cometd
- camel-ftp
- camel-http & camel-http4
- camel-http4
- camel-irc

- camel-jetty8 & camel-jetty9
- camel-linkedin
- camel-mail
- camel-mina2
- camel-netty & camel-netty4
- camel-olingo2
- 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` – this is 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 would look like:

```
<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 we have a top-level `sslContextParameters` element and then `keyManagers` and `trustManagers` children. These respectively configure the key store and trust store files that we created back 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?sslContextParametersRef=sslContextParameters"/>
    <transform>
        <constant>Hi</constant>
    </transform>
</route>
```

With this route you now have a HTTPS endpoint up and running. In Java, things get a bit more verbose, but it is essentially 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("sslContextParameters", sslContextParameters);

return registry;
}

```

The Java route is also similar:

```

from("jetty:https://localhost:8080/early?sslContextParametersRef=sslContextParameters")
.transform().constant("Hi");

```

When calling these HTTPS endpoints within Camel you'll need to also provide the `sslContextParameters` that contains a trusted certificate. For the purposes of the example we can just reuse the server `sslContextParameters`. The URI syntax is exactly same for the producer in this case:

```

@Test
public void testHttps() throws Exception {
    String reply =
        template.requestBody("jetty:https://localhost:8080/early?sslContextParametersRef=sslCo
        ntextParameters", "Hi Camel!", String.class);
    assertEquals("Hi", reply);
}

```

Of course, if we didn't provide any `sslContextParameters` with a valid trust store we should expect Camel to throw an execution exception as the server wouldn't let us 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 out for yourself using the following Maven goals from the chapter14/transport directory:

```

mvn test -Dtest=HttpsTest
mvn test -Dtest=SpringHttpsTest

```

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 certainly always go together. Authentication is the first step whereby a user proves who they are to a system using typically a username and password. Next, after successful authentication, authorization is the process where the system checks what you are 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, Spring Security. Both of these frameworks provide a means of authenticating and authorizing a user. They do more than just this of course but from a Camel point-of-view, we are just utilizing 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'll 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'll 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>2.16.0</version>
</dependency>
```

This will bring in the necessary Spring Security JARs transitively so you don't need to add them to your POM as well. If you are curious though, or are looking at the reference documentation, Camel uses the spring-security-core and spring-security-config modules. There are several more modules 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 at <http://projects.spring.io/spring-security>.

Route security in Camel is handled via a "policy" - classes which implement org.apache.camel.spi.Policy. A route protected by a policy looks like:

```
<route>
  <from uri="direct:start"/>
  <to uri="mock:unsecure"/>
  <policy ref="admin">
    <to uri="mock:secure"/>
  </policy>
</route>
```

Here we are referencing a policy defined by the "admin" bean. Note that the whole route is not protected by the policy but only what is inside the policy element. So you must take care to place the secured calls inside the policy element. Policy is a generic Camel concept so we

haven't really done any securing yet. Let's take a look at that "admin" bean to see how it is helping to secure our route:

```
<authorizationPolicy id="admin" access="ROLE_ADMIN"
    authenticationManager="authenticationManager"
    accessDecisionManager="accessDecisionManager"
    xmlns="http://camel.apache.org/schema/spring-security"/>
```

This `authorizationPolicy` element is basically syntactic sugar for the `org.apache.camel.component.spring.security.SpringSecurityAuthorizationPolicy` class, which implements the `Policy` interface. So we could have created a `SpringSecurityAuthorizationPolicy` bean here just the same. Let's go over what the attributes of `authroizationPolicy` are doing. First, the "`accessDecisionManager`" attribute specifies the `org.springframework.security.access.AccessDecisionManager` that we will 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 (i.e. A String beginning with prefix "ROLE_"). Lastly, the "`authenticationManager`" attribute specifies the `org.springframework.security.authentication.AuthenticationManager` bean that we'll be using to authenticate the user.

CONFIGURING SPRING SECURITY

Outside Camel, there are 2 required Spring Security beans that you need to configure: the `AccessDecisionManager` and the `AuthenticationManager`. To configure these you will 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 we can then define the `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 most simple `AuthenticationManager` configuration you can get. All we 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. Of course, it is far easier to talk about an XML snippet over an LDAP server configuration! In the snippet above, we are defining two users: jon and claus. Then both have a different set of roles. Only claus has the admin role. Recall before that our 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 the AccessDecisionManager. For our role checking example, we can use the following AccessDecisionManager configuration:

```
<bean id="accessDecisionManager"
    class="org.springframework.security.access.vote.AffirmativeBased">
    <property name="decisionVoters">
        <list>
            <bean class="org.springframework.security.access.vote.RoleVoter"/>
        </list>
    </property>
</bean>
```

The concrete AccessDecisionMaker we are using here, AffirmativeBased, will grant access if one of the decision voters succeeds. We only have one decision voter configured here so it is basically 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 is ROLE_ADMIN).

So now our route is secured! However, we need to do a little more work to actually use it now. Trying to access it at this point will certainly fail unless we pass in the proper credentials. The SpringSecurityAuthorizationPolicy looks for Exchange.AUTHENTICATION headers (i.e. "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("Davs 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 we are sending two messages: one using claus's credentials and another using jon's. Of course since our policy requires the "admin" role. Only claus's message gets through. On sending jon's credentials, Camel throws a CamelExecutionException with cause being a 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 please see the Camel website <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, security can be implemented in many different 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, next up we'll be talking all about running and deploying Camel in various environments.

15

Running and deploying Camel

This chapter covers

- Starting and stopping Camel safely
- Adding and removing routes to existing Camel
- Where you can run Camel?
- Running standalone
- Running in web containers
- Running in Java EE servers
- Running with OSGi
- Running with CDI

In the previous chapter, 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 start with the topic of running Camel—you'll 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 how Camel and routes are started. We'll continue on this path, looking at how you can dynamically start and stop routes at runtime. Your applications won't run forever, so we'll spend some time focusing on how to shut down Camel in a safe manner.

The other part of the chapter covers various strategies for deploying Camel. We'll take a look at five common runtime environments supported by Camel.

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 13.1 shows a high-level diagram of the application.

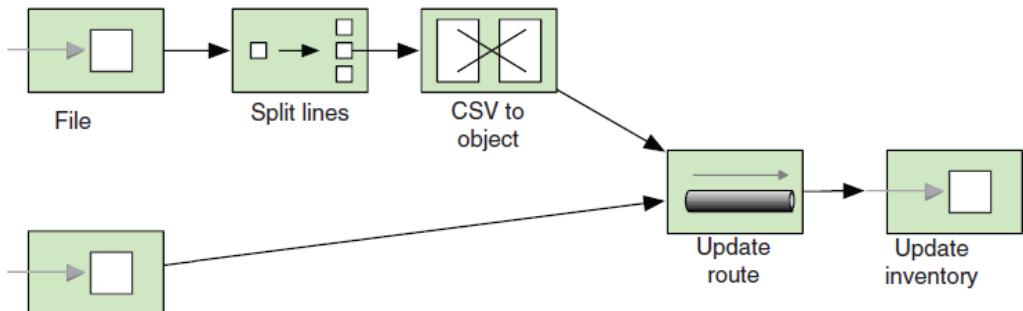


Figure 15.1 A Rider Auto Parts application accepting incoming inventory updates from either files or a web service

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 order in which the routes are started 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 now 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 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.

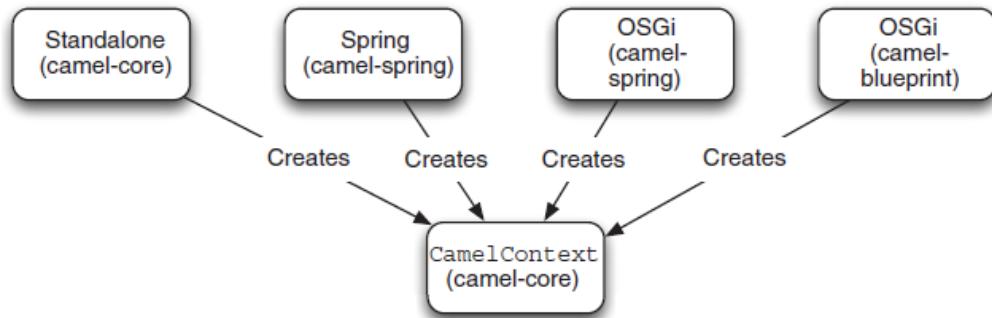


Figure 15.2 Using Camel with containers often requires the container in question to prepare and create CamelContext up front before it can be started. TODO update for CDI

Because Spring is a common container, we'll outline how Spring and Camel work together to prepare a 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 a `org.springframework.beans.factory.xml.NamespaceHandler`, which is the extension point for using custom namespaces in Spring XML files. Camel provides the `CamelNamespaceHandler`.

When using Camel in the Spring XML file, you would 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, it must be identified in the `META-INF/spring.handlers`, where you map the namespace to the class implementation:

```
http://camel.apache.org/schema/spring=
org.apache.camel.spring.handler.CamelNamespaceHandler
```

The `CamelNamespaceHandler` is then responsible for parsing the XML and delegating to other factories for further processing. One of these factories is the `CamelContextFactoryBean`, which is responsible for creating the `CamelContext` that essentially is your Camel application.

When Spring is finished initializing, it signals to third-party frameworks that they can start by broadcasting the `ContextRefreshedEvent` event.

STARTING CAMELCONTEXT

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.

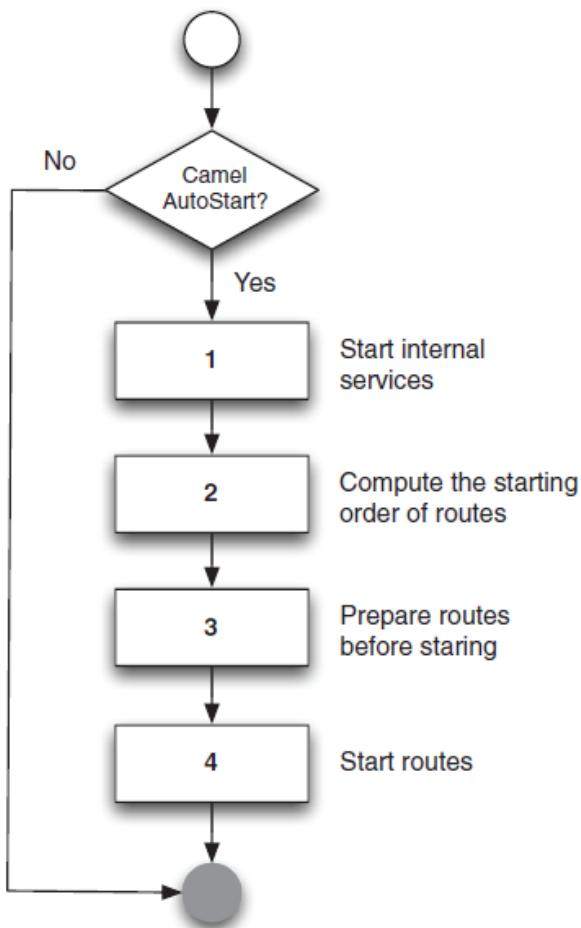


Figure 15.3 Flow diagram showing how Camel starts by starting internal services, computing the starting order of routes, and preparing and starting the routes.

`CamelContext` is started by invoking its `start` method. The first step in figure 15.3 determines whether or not autostartup is enabled for Camel. If it's disabled, the entire startup process is skipped. By default, Camel is set to autostart, which involves 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 will start up all the routes in the order they are defined in the Spring XML files or the RouteBuilder classes. We'll cover how to configure the order of routes in section 15.1.3.
3. Prepare routes—Prepares the routes before they're started.
4. Start routes—Starts the routes by starting the consumers, which essentially 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 has been started and that the startup process is complete.

In some cases, you may need to influence how Camel is started, and we'll look at that now.

15.1.2 Camel startup options

Camel offers various options when it comes to starting Camel. For example, you may have a maintenance route that should not 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.

The options from table 15.1 can be divided into two kinds. The first four options are related to startup and shutdown, and the remainder are miscellaneous options. We'll look at how to use the miscellaneous options first, and then we'll turn our attention to the startup and shutdown options.

Table 15.1 Camel startup options

| Option | Description |
|---------------------|--|
| AutoStartup | This option is used to indicate whether or not the route should be started automatically when Camel starts. This option is enabled by default. |
| StartupOrder | This option dictates the order in which the routes should be started when Camel starts. We'll cover this in section 15.1.3. |
| ShutdownRoute | This option is used to configure whether or not the route in question should stop immediately or defer while Camel is shutting down. We'll cover shutdown in section 15.3. |
| ShutdownRunningTask | This option is used to control whether Camel should continue to complete pending running tasks at shutdown or stop immediately after the current task is complete. We'll cover shutdown in section 15.3. |
| Tracing | This option is used to trace how an exchange is being routed within that particular route. This option is disabled by default. |

| | |
|----------------|---|
| Delayer | This option is used to set a delay in milliseconds that slows down 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 | This option is used to store the history of an exchange as its being routed within a particular route. This option is enabled by default. |
| HandleFault | This option is used to turn fault messages into exceptions. This is not a typical thing to do in a pure Camel application, but when deployed into a JBI container like Apache ServiceMix, you'll need to set this option to let the Camel error handler react to faults. This option is disabled by default. We'll cover this in more detail shortly. |
| StreamCaching | This option is used to cache 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'll cover this in more detail shortly. |

CONFIGURING STREAMCACHING

The miscellaneous options are often used during development to turn on additional logging, such as the Tracing option, which we'll 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("jbi:service:http://rider.com/AutoPartService")
            .to("xslt:html-parts.xsl")
            .to("jbi:service:http://rider.com/HtmlService");
    }
}
```

The same example using Spring XML would look like this:

```
<camelContext streamCache="true"
    xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="jbi:service:http://rider.com/AutoPartService"/>
        <to uri="xslt:html-parts.xsl"/>
        <to uri="jbi:service:http://rider.com/HtmlService"/>
    </route>
</camelContext>
```

All the options from table 15.1 can be scoped at either context or route level. The preceding stream cache example was scoped at context level. You could also configure it on a particular route:

```
public class MyRoute extends RouteBuilder {
    public void configure() throws Exception {
```

```

        from("jbi:service:http://rider.com/AutoPartService")
            .streamCaching()
            .to("xslt:html-parts.xsl")
            .to("jbi:service:http://rider.com/HtmlService");
    }
}

```

You can configure route-scoped stream caching in Spring XML as follows:

```

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route streamCache="true">
        <from uri="jbi:service:http://rider.com/AutoPartService"/>
        <to uri="xslt:html-parts.xsl"/>
        <to uri="jbi:service:http://rider.com/HtmlService"/>
    </route>
</camelContext>

```

NOTE Java DSL uses the syntax `noXXX` to disable an option, such as `noStreamCaching` or `noTracing`.

There is one last detail to know 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 route messages using the JBI component. The `HandleFault` option is used to control whether or not Camel error handling should react to faults.

Suppose sending to the `jbi:service:http://rider.com/HtmlService` 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 you can 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("jbi:service:http://rider.com/AutoPartService")
            .streamCaching().handleFault()
            .to("xslt:html-parts.xsl")
            .to("jbi:service:http://rider.com/HtmlService");
    }
}

```

The equivalent example in Spring 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="jbi:service:http://rider.com/AutoPartService"/>

```

```

<to uri="xslt:html-parts.xsl"/>
<to uri="jbi:service:http://rider.com/HtmlService"/>
</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 becomes more and more important the more interdependent the routes are. For example, you may have reusable routes that must be started before being leveraged 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 or not. 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.

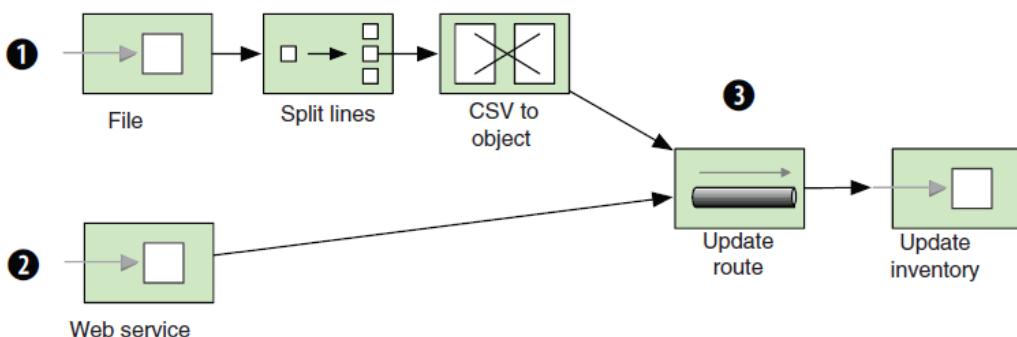


Figure 15.4 Camel application with two input routes ① and ② which depend on a common route ③

- ① The file-based route will poll incoming files and split each line in the file. The lines are then converted to an internal `camelinaction.inventory.UpdateInventoryInput` object, which is sent to the ③ route.
- ② The web service route is much simpler because incoming messages are automatically converted to the `UpdateInventoryInput` object. The web service endpoint is configured to do this.
- ③ This route 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 upon route ③, and that's why you need to 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("cxft: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 `StartupOrder`. 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 scheme `route1`, `route2`, and so forth.

Listing 15.2 Spring XML version of listing 15.1

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="webservice" startupOrder="3">
        <from uri="cxft: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="camelinaction.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>
```

You should 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 STARTUP ORDER 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 have used the numbers 5, 20, and 87 instead of 1, 2, and 3. All that matters is that the numbers must be unique.

You can also omit assigning a `StartupOrder` to some of the routes. In that case, Camel will assign these routes a unique number starting with 1,000 upwards. This means that the numbers from 1 to 999 are free 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. It's only important when you have route dependencies, as in the previous example.

The source code for the book contains this example in the `chapter15/startup` directory. You can try it out 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 you can 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 or not 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 13.1. 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 how we used `noAutoStartup()` to disable route startup. We could also have used `autoStartup(false)`, `autoStartup("false")`, or even `autoStartup("${{startupRouteProperty}}")`, where a property is referenced. In the XML DSL there is 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 by 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. In the next section, we'll look at various ways of programmatically controlling the lifecycle of routes at runtime.

15.2 Starting and stopping routes at runtime

In chapter 16, you will 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. In this section, we'll look at how to do this.

You can start and stop routes at runtime in several ways, including these:

- *Using CamelContext*—By invoking the `startRoute` 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 have remote management enabled, you can control the routes from another machine.
- *Using the ControlBus component*—Another way of controlling routes is by using the Control Bus EIP. Yes, it's actually an enterprise integration pattern! Camel implements

this via the ControlBus component, instead of using a method in the DSL. The control bus allows you to manage routes at runtime using regular messaging instead of some special SPI. By sending a message to a ControlBus endpoint, you can start, stop, resume, or suspend a route. This is discussed in the next chapter in section 16.4.4.

The use of JMX was covered in the previous chapter, so we'll discuss using `CamelContext`, `ControlBus` and `RoutePolicy` in this chapter.

15.2.1 Using CamelContext to start and stop routes at runtime

The `CamelContext` provides methods to easily start and stop routes.

To illustrates 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 forgot 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 are 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 only pick up 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 the `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 there is an exchange in progress. It's also important to call `stopRoute` from a thread separate to 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 source code for the book 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 was simple, using the inlined processor at the end of the route isn't an optimal solution. It would be better to keep the business logic separated from the stopping logic. This can be done with a feature called `OnCompletion`.

USING ONCOMPLETION

`OnCompletion` is a feature that allows you to do *additional* routing after the original route is done. The classic example would be to send an email alert if a route fails, but it has a broad range of uses.

Instead of using the inlined processor to stop the route, you can use `OnCompletion` in the `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 the `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(exchange, "manual");
        exchange.getContext().stopRoute(name);
    }
}
```

This improves the readability of the route, as it's shorter and doesn't mix high-level routing logic with low-level implementation logic. By using `OnCompletion`, the stopping logic has been separated from the original route.

Scopes can be used to define `OnCompletions` at different levels. Camel supports two scopes: context scope (high level) and route scope (low level). In the preceding example, you used context scope. If you wanted to use route scope, you'd 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 only trigger when the route completes successfully or when it fails by using the `OnCompleteOnly` or `OnFailureOnly` options. For example, you can use `OnFailureOnly` to build a route that sends an alert email to support personnel when a route fails.

The source code for the book contains this example in the chapter15/startup directory. You can try it using the following Maven goal:

```
mvn test -Dtest=ManualRouteWithOnCompletionTest
```

We've now covered how to stop a route at runtime using the `CamelContext` API. We'll now look at another feature called `RoutePolicy`, which can also be used to control the lifecycle of routes at runtime.

15.2.2 Using RoutePolicy to start and stop routes at runtime

A `RoutePolicy` is a policy that can control routes at runtime. For example, a `RoutePolicy` can control whether or not a route should be active. But you aren't limited to such scenarios—you can implement any kind of logic you wish.

The `org.apache.camel.spi.RoutePolicy` is an interface that defines several callback methods 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, etc. You're free to implement whatever logic you want in these callbacks. For convenience, Camel provides the `org.apache.camel.impl.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.

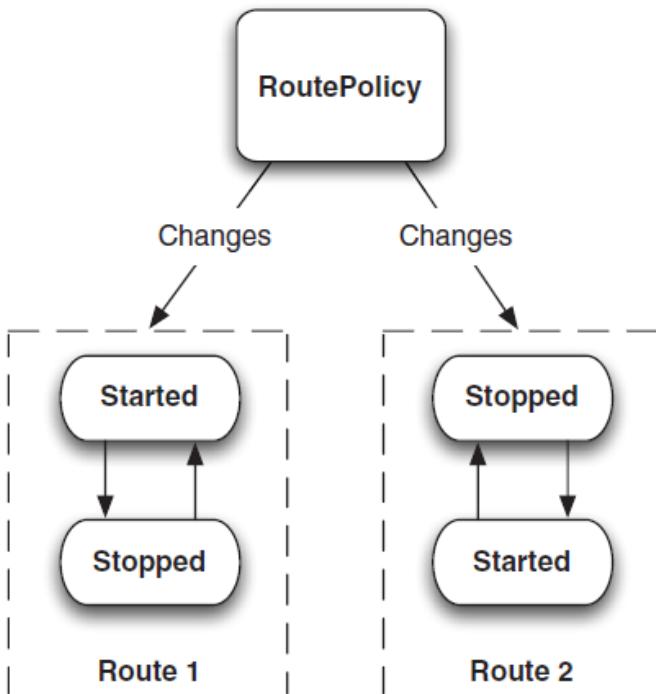


Figure 15.5 `RoutePolicy` changes the active state between the two routes so only one route is active at any time.

As you can see in this figure, the `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) { ❶
        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(exchange);
            context.stopRoute(stop); ❷
            context.startRoute(start);
        } catch (Exception e) {
            getExceptionHandler().handleException(e);
        }
    }
}
```

❶ Identifies routes to flip

❷ Flips the two routes

In the constructor, you identify the names of the two routes to flip ❶. As you extend the `RoutePolicySupport` class, you only override 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 then use `CamelContext` to flip the routes ❷. If an exception is thrown, you let the `ExceptionHandler` take care of it, which by default will log 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 Spring XML, 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 source code for the book 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 there may be numerous in-flight messages being processed. Shutting down while messages are in flight may harm your business because those messages could potentially be lost. So the goal of shutting down a Camel application reliably is to shut it down when its *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 2.18.0 (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 2.18.0 (CamelContext: camel-1) uptime 0.684 seconds
SpringCamelContext      - Apache Camel 2.18.0 (CamelContext: camel-1) is shutdown in 0.025
seconds
```

This tells you a few things. You can see the graceful shutdown is using a 10-second timeout. 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 actually 300 seconds but the log was a unit test and `CamelTestSupport` uses just 10 seconds. You can configure this timeout yourself on the `CamelContext`. For example, to use 20 seconds as the default timeout value, you can do as follows:

```
camelContext.getShutdownStrategy().setTimeout(20);
```

Doing this in Spring XML requires a bit more work, because you have to define a Spring bean to set the timeout value:

```
<bean id="shutdown" class="org.apache.camel.impl.DefaultShutdownStrategy">
    <property name="timeout" value="20"/>
</bean>
```

Notice that the timeout value is in seconds.

Then Camel logs the progress of the routes as they shut down, one by one, according to the order in which they were started. Notice that the `file` route is suspended and deferred, and then later is shut down.

This is a little glimpse of the complex logic the graceful shutdown process uses to shut down Camel in a reliable manner. We'll cover what *suspension* and *defer* mean in a moment.

At the end, Camel logs the completion of the graceful shutdown, which in this case was really fast and completed in less than one second. Camel also logs whether there were any in-flight messages just before it stops completely.

If there were in-flight exchanges holding up the graceful shutdown, Camel would print a status about these every second like:

```
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 would continue until the in-flight exchanges complete or the timeout is reached. In the case of a timeout, this means Camel did not complete the graceful shutdown and it would log a warning showing what in-flight exchanges 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]
```

TIP All this additional logging may not be desirable for all applications. Maybe it is expected that on shutdown messages are simply discarded. In such cases you can disable the additional logging. To suppress the warning on timeout you can do the following:

```
context.getShutdownStrategy().setSuppressLoggingOnTimeout(true);
```

and for suppressing the status messages about in-flight exchanges during graceful shutdown you can do:

```
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 wanted to test what happens if you shut down Camel while it was working on the big file. Let's take a look at how Camel handles this.

At first, you see the usual logging about the shutdown in progress:

```
SpringCamelContext
  - Apache Camel 2.18.0 (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 is a log line indicating that Camel has noticed in-flight exchanges, which is 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.
update = 1]
```

```
Inflights per route: [file = 2,
```

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 2.18.0 (CamelContext: camel-1) uptime 17.747 seconds
SpringCamelContext
  - Apache Camel 2.18.0 (CamelContext: camel-1) is shutdown in 14.028 seconds
```

Notice how the routes were shutdown in the reverse order that they were started in. 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 Spring XML is as follows:

```
<route id="update" startupOrder="1" shutdownRoute="Defer"

```

The source code for the book contains this example in the chapter15/shutdown directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=GracefulShutdownBigFileTest
mvn test -Dtest=GracefulShutdownBigFileXmlTest
mvn test -Dtest=GracefulShutdownBigFileDeferTest
mvn test -Dtest=GracefulShutdownBigFileDeferXmlTest
```

As you’ve just learned, Camel end users are responsible for configuring routes correctly to support reliable shutdown. Some may say this is a trade-off, but we, the Camel team, think of it as flexibility and don’t believe computer logic can substitute for human logic in ensuring a reliable shutdown. We think it’s best to give Camel users the power to configure their routes to support their use cases.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

ABOUT STOPPING AND SHUTTING DOWN

Camel will leverage the graceful shutdown mechanism when it stops or shuts down routes. That means the example in 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 will also unregister the route from management (JMX). Use `stopRoute` when you want to be able to start the route again. Only use `shutdownRoute` 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 deployment strategies that are possible.

15.4 Deploying Camel

Camel is described as a lightweight and embeddable integration framework. This means that it supports more deployment strategies and flexibility than traditional ESBs and application servers. Camel can be used in a wide range of runtime environments, from standalone Java applications, to web containers, to the cloud.

In this section, we'll look at four different deployment strategies that are possible with Camel and present their strengths and weaknesses.

- Embedding Camel in a Java application
- Running Camel in a web environment on 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

These five deployment strategies are the common ways of deploying Camel.

15.4.1 Embedded in a Java application

It's appealing to embed Camel in a Java application 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.

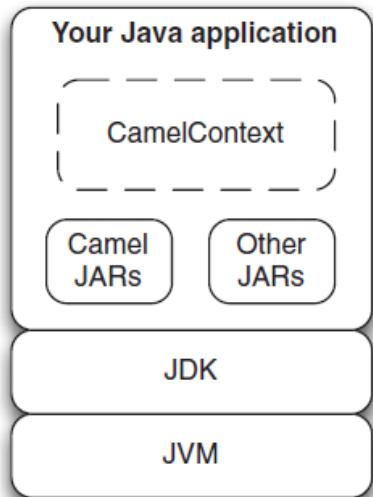


Figure 15.6 Camel embedded in a standalone Java application

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 discussed 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 a `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 the `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 only shut down 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 just need to start 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 an `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 leverage instead of writing your own class. You can change the previous `InventoryMain` class to leverage 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.start();
    }
}
```

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 obviously also stop the Camel application by invoking the `stop` method on the `main` instance.

The source code for the book contains this example in the `chapter15/standalone` directory. You can try it out 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 RouterBulder class, Spring XML file should be loaded. By default, the `org.apache.camel.spring.Main` class will load 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 • Works well with thick client applications, such as a Swing or Eclipse rich client GUI | <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 leverage 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 were 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 have existing infrastructure to be used 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 will be done automatically.

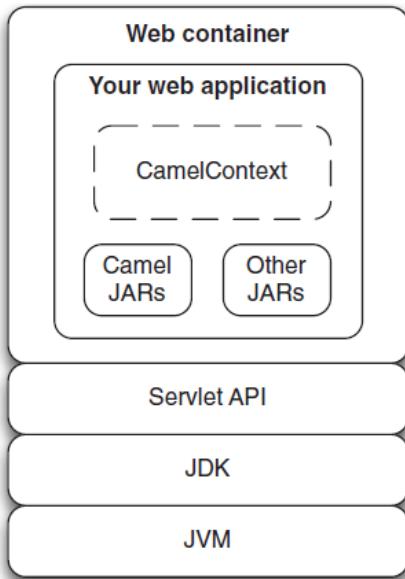


Figure 15.7 Camel embedded in a web application

The Camel instance embedded in your web application is bootstrapped by Spring. By leveraging 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 only need 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_2_5.xsd"
          version="2.5">
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
</web-app>
  
```

This context listener also takes care of shutting down Camel properly 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'll need to add a context parameter which 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   <bean id="inventoryService" class="camelaction.InventoryService"/>
    <bean id="inventoryRoute" class="camelaction.InventoryRoute"/>
    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <routeBuilder ref="inventoryRoute"/>
    </camelContext>
</beans>
```

① Imports CXF from another XML file

Listing 15.6 is a regular Spring XML file in which you can use the `<import>` tag to import other XML files. For example, this is done by having CXF defined in the camel-cxf.xml file ① . Camel, itself, is embedded using the `<camelContext>` tag.

The source code for the book contains this example in the chapter15/war directory. You can try it out by 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>9.2.19.v20160908</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 2.18.0 (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 how you can run 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 leverage the hot deployment of Apache Tomcat, so you must first start it. Here's how you can start it on a Linux system using the `bin/startup.sh` script:

```
[janstey@ghost apache-tomcat-8.5.8]$ bin/startup.sh
Using CATALINA_BASE:  /home/janstey/kits/apache-tomcat-8.5.8
Using CATALINA_HOME:  /home/janstey/kits/apache-tomcat-8.5.8
Using CATALINA_TMPDIR: /home/janstey/kits/apache-tomcat-8.5.8/temp
Using JRE_HOME:        /usr/java/jdk1.8.0_91/
Using CLASSPATH:       /home/janstey/kits/apache-tomcat-
                      8.5.8/bin/bootstrap.jar:/home/janstey/kits/apache-tomcat-8.5.8/bin/tomcat-juli.jar
Tomcat started.
```

This will start Tomcat in the background, so you need to tail the log file to see what happens:

```
[janstey@ghost apache-tomcat-8.5.8]$ 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-8.5.8/webapps/
```

Then 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 2.18.0 (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. This can be done 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>.

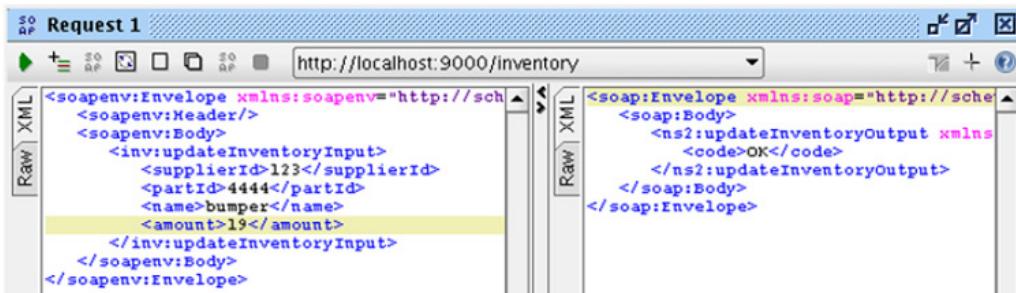


Figure 15.8 Using SoapUI testing the web service from the deployed application in Apache Tomcat

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
```

There’s another great benefit of this deployment model, which 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 leverage this 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 leverage 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.

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_2_5.xsd">
```

```

        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        version="2.5">
<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 upon the HTTP transport instead of Jetty, as follows:

```

<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http</artifactId>
    <version>${cxf-version}</version>
</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

```

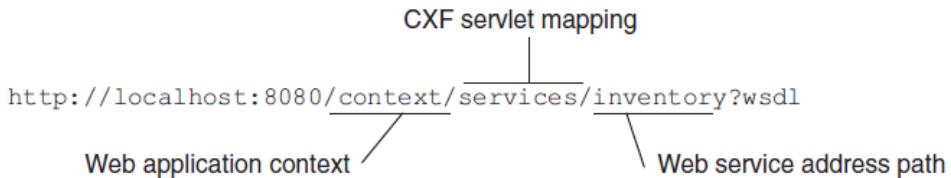
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxf="http://camel.apache.org/schema/cxf"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        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
    <cxf:cxfEndpoint id="inventoryEndpoint" ②
        address="/inventory"
        serviceClass="camelaction.inventory.InventoryEndpoint"/>
</beans>

```

- ① Required import when using servlet
- ② Web service endpoint address

To use Apache CXF in a servlet container, you have to import the cxf-servlet.xml resource ① . This exposes the web service via the servlet container, which means the endpoint address has to be adjusted to a relative context path ② .

In the previous example, the web service was available at `http://localhost:9000/inventory?wsdl`. By using Apache Tomcat, the web service is now exposed at this address:



Notice that the TCP port is 8080, which is the default Apache Tomcat setting.

The source code for the book 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 (WildFly 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 we discussed in the previous section. But WildFly has a slightly different classloading mechanism, so you need to take care when loading things like type converters. If you are using the `came-bindy` component you will even have to leverage a special Camel JBoss component to adapt to this classloading. This component isn't provided out of the box at 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'll 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. So, for our single type converter we need to tell Camel the fully qualified name of the type converter class to load. So we make a change to src/main/resources/META-INF/services/org/apache/camel/TypeConverter like:

```
-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 10.1.0.Final is started as follows:

```
[janstey@ghost wildfly-10.1.0.Final]$ ./bin/standalone.sh
```

After a couple seconds, WildFly is ready, and this is logged to the console:

```
20:24:18,973 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: WildFly Full
10.1.0.Final (WildFly Core 2.2.0.Final) started in 2024ms - Started 331 of 577
services (393 services are lazy, passive or on-demand)
```

Then the WAR file is copied:

```
[janstey@ghost war-jboss]$ cp target/riderautoparts-war-wildfly-2.0.0.war ~/kits/wildfly-
10.1.0.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 leverage 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 camel-bindy component and also use the fully qualified class name when loading type converters.

THE WILDFLY-CAMEL SUBSYSTEM

There is another way to deploy Camel applications to WildFly which provides a much tighter integration between Camel and WildFly. This is by using a project called the WildFly-Camel Subsystem⁶. 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 3rd party libraries that come with a Camel component may conflict with what is included in the WildFly container. WildFly Camel takes care of this dependency management for you so you can just focus on the actual Camel application.

You can even write Camel routes directly in WildFly XML configuration like:

```
<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 this out, you'll first need to install WildFly Camel as it's 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 4.4.0, so we used that.

After installing WildFly Camel you can boot WildFly up with Camel support by running:

```
./bin/standalone.sh -c standalone-full-camel.xml
```

Now, if standalone/configuration/standalone-full-camel.xml contained a camelContext element as above, it would be started with the container and you'd see something like this in the logs:

⁶ Check out the project source at <https://github.com/wildfly-extras/wildfly-camel> and docs at <https://wildflyext.gitbooks.io/wildfly-camel/content>

```

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 2.18.1 (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 separate 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, in the previous WAR example we had a CamelContext defined in `src/main/webapp/WEB-INF/applicationContext.xml`. WildFly-Camel will search in META-INF so we 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:

```

[janstey@ghost chapter15]$ ls -lah wildfly-war/target/riderautoparts-war-wildfly-2.0.0.war
-rw-rw-r--. 1 janstey janstey 15M Dec  8 20:23 war-jboss/target/riderautoparts-war-wildfly-
  2.0.0.war
[janstey@ghost chapter15]$ ls -lah wildfly-camel-war/target/riderautoparts-wildfly-camel-war-
  2.0.0.war
-rw-rw-r--. 1 janstey janstey 16K Jan  8 18:20 wildfly-camel-war/target/riderautoparts-
  wildfly-camel-war-2.0.0.war

```

To ensure you end up with this skinny WAR file, you must use the provided scope for all Camel dependencies in your POM. So for our example we have dependencies like:

```

<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>

<!-- using http servlet transports -->
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http</artifactId>
  <version>${cxf-version}</version>
  <scope>provided</scope>
</dependency>

```

To run deploy this example to a locally running WildFly instance, go to the chapter15/wildfly-camel-war directory of the books source and run:

```
mvn clean install -Pdeploy
```

This will build a skinny WAR with our inventory-camel-context.xml and supporting Java files and then deploy to WildFly.

You can also avoid XML configuration altogether by simply 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 leverage 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"> • 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 where 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 will show 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 actual OSGi container. The example presented here is included with the source code for the book in the chapter15/osgi directory.

NOTE In this book, we won'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, Karl Pauls, Stuart McCulloch, and David Savage (Manning). 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. 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>
    <version>${bundle-plugin-version}</version>
    <extensions>true</extensions>
    <configuration>
      <instructions>
        <Bundle-Name>${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 the `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 source code for the book 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 have 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 4.0.8.) 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)

This should start up Karaf and display a logo when it's ready like:

```
[janstey@ghost apache-karaf-4.0.8]$ ./bin/karaf

      _/\_/\_/\_/\_
     / ,< / \_` / \_/\_/\_/\_
    / | | / / \_ / / / / / \_ / /
   /_ \_| \_,/_/ \_,/_/ \_,/_/ \_/

Apache Karaf (4.0.8)

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 by 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 2.18.1
```

You can then type `feature:list` to see all the features that are available to be installed. Among these should be several Camel-related features. To narrow the list you can use grep just like you would on the command line:

```
feature:list | grep camel
```

The example application requires the `http`, `camel`, and `camel-cxf` features. Type this command in the shell:

```
feature:install http camel camel-cxf
```

And then wait a bit.

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 a Blueprint-based Camel route

The most popular way to deploy Camel routes to an OSGi container like Apache Karaf is in a blueprint XML file. Blueprint itself 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 simply 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 off, the XML file is stored under the `OSGI-INF/blueprint` directory, which is different than Spring's `META-INF/spring`. Within the file it looks very 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 it's 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 can easily be done 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:

```
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 | 2.18.1 | camel-blueprint
58 | Active  | 50 | 2.18.1 | camel-catalog
59 | Active  | 50 | 2.18.1 | camel-commands-core
60 | Active  | 50 | 2.18.1 | camel-core
61 | Active  | 50 | 2.18.1 | camel-cxf
62 | Active  | 50 | 2.18.1 | camel-cxf-transport
63 | Active  | 50 | 2.18.1 | camel-spring
64 | Active  | 80 | 2.18.1 | 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 an `bundle:list` again:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/camel-in-action-second-edition>

Licensed to Ivan Prochazka <ivanp@hispeed.ch>

```
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`.

So how can you test that it works? You can 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 - 2.18.1
| Apache Camel 2.18.1 (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 how we can use 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 only vary by a few parameters. Going back to our Rider Auto Parts example, let's say we needed to spin up several instances of the file inventory route, with each instance consuming from a different directory. Since 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 would be to have a template route that accepts a few parameters, and then have a MSF spin up a new instance based on configuration updates. For example, if we extract the file consumer route out into its own template class we'd have a reusable route like:

```
public class FileInventoryRoute extends RouteBuilder {

    private String inputPath;
    private String routeId;

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

```

        .split(body().tokenize("\n"))
        .convertBodyTo(UpdateInventoryInput.class)
        .to("direct:update");
    }

    public String getInputPath() {
        return inputPath;
    }

    public void setInputPath(String inputPath) {
        this.inputPath = inputPath;
    }

    public String getRouteId() {
        return routeId;
    }

    public void setRouteId(String routeId) {
        this.routeId = routeId;
    }
}

```

So before this route would be useful, you'd have to set the `inputPath` and `routeId` fields of course. Looking at our blueprint XML file from the previous section, we just have to add an extra element:

```

<bean id="camelServiceFactory" class="camelinaction.FileInventoryServiceFactory" init-
    method="init">
    <property name="bundleContext" ref="blueprintBundleContext"/>
    <property name="camelContext" ref="myCamelContext"/>
</bean>

```

This will start up our MSF `FileInventoryServiceFactory`, which can spin up additional instances of our `FileInventoryRoute` template route. The service factory is shown in listing 15.9.

Listing 15.9 A managed service factory for spinning up `FileInventoryRoute` instances

```

package camelinaction;

import java.util.Dictionary;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

import org.apache.camel.CamelContext;
import org.osgi.framework.BundleContext;
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceRegistration;
import org.osgi.service.cm.ConfigurationException;
import org.osgi.service.cm.ManagedServiceFactory;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class FileInventoryServiceFactory
    implements ManagedServiceFactory {

```

①

```

private static final Logger LOG =
    LoggerFactory.getLogger(FileInventoryServiceFactory.class);

private CamelContext camelContext;
private BundleContext bundleContext;
private Map<String, FileInventoryRoute> routes
    = new HashMap<String, FileInventoryRoute>();      ②
private ServiceRegistration registration;

@Override
public String getName() {
    return "FileInventoryRouteCamelServiceFactory";
}

@SuppressWarnings("unchecked")
public void init() {
    Dictionary properties = new Properties();
    properties.put( Constants.SERVICE_PID,
        "camelaction.fileinventoryroutefactory");      ③
    registration = bundleContext.registerService(
        ManagedServiceFactory.class.getName(),
        this, properties);      ③

    LOG.info("FileInventoryRouteCamelServiceFactory ready to accept new config with
PID=camelaction.fileinventoryroutefactory-xxx");
}      ③

@Override
public void updated(String pid,          ④
    Dictionary<String, ?> properties)      ④
throws ConfigurationException {          ④

    String path = (String) properties.get("path");      ⑤

    LOG.info("Updating route for PID=" + pid + " with new path=" + path);

    // need to remove old route before updating
    deleted(pid);      ⑥

    // now we create a new route with update path
    FileInventoryRoute newRoute = new FileInventoryRoute();
    newRoute.setInputPath(path);
    newRoute.setRouteId("file-" + pid);

    // finally we add the route
    try {
        camelContext.addRoutes(newRoute);      ⑦
    } catch (Exception e) {
        LOG.error("Failed to add route", e);
    }
    routes.put(pid, newRoute);
}

@Override
public void deleted(String pid) {          ⑧
    LOG.info("Deleting route with PID=" + pid);
}

```

```

        try {
            FileInventoryRoute route = routes.get(pid);
            if (route != null) {
                camelContext.stopRoute(route.getId());
                camelContext.removeRoute(route.getId());
                routes.remove(pid);
            }
        } catch (Exception e) {
            LOG.error("Failed to remove route", e);
        }
    }

    ...
}

```

- ➊ A service factory must implement org.osgi.service.cm.ManagedServiceFactory
- ➋ A map between persistent identities (PIDs) and routes
- ➌ Register the service factory and provide the factory PID to watch for configuration
- ➍ The updated method is called when configuration changes
- ➎ The sole configuration property we will be using for route instances
- ➏ Need to remove the old route before adding the updated one
- ➐ Add the new route to the CamelContext
- ➑ The deleted method is called when the corresponding configuration is deleted

To explain what's going on here you need to know about how configuration works in an OSGi container like 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 we assign a factory PID "camelaction.fileinventoryroutefactory" which means Configuration Admin will watch for configuration with PIDs like "camelaction.fileinventoryroutefactory-foo", "camelaction.fileinventoryroutefactory-bar", etc. They just have to start with the factory PID for our MSF to get picked up. To put it more concretely, in the Karaf distribution, if we add a camelaction.fileinventoryroutefactory-path1.cfg file to the etc directory with contents:

```
path=file://target/inventory/updates
```

This 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 the CamelContext. In the logs you'd see something like:

```

2016-12-21 19:31:05,127
| INFO
| f7-466f8727ed59)
| BlueprintCamelContext
| 60 - org.apache.camel.camel-core - 2.18.1
| Route: file-camelaction.fileinventoryroutefactory.32f4ffa3-e2cc-4930-a9f7-
466f8727ed59 started and consuming from: file://target/inventory/updates

```

We could add as many extra configuration files as we 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.

You've now seen how to deploy a Camel application into an OSGi container. Table 13.5 lists the pros and cons of deploying Camel in an OSGi container.

Table 13.5 Pros and cons of using OSGi as a deployment strategy

| Pros | Cons |
|---|--|
| <ul style="list-style-type: none"> • Leverages OSGi for modularity • Provides classloader isolation and hot deployment • Commitment in the open source community and from big vendors endorsing OSGi | <ul style="list-style-type: none"> • Involves a learning curve for OSGi • Unsupported third-party frameworks; many frameworks have yet to become OSGi compliant • Requires extra effort to decide what package imports and exports to use for your module |

Even after looking into a more advanced MSF style deployment, we've really only scratched the surface of OSGi in this chapter. If you go down that path, you'll need to pick up some other books, because OSGi is a big concept to grasp and master. It's also very powerful and allows you to create very dynamic applications. On the other hand, the path of the web application is the beaten track, and there are plenty of materials and people who can help you if you come up against any problems.

15.6 Camel and CDI

Contexts and Dependency Injection a.k.a. CDI is the part of the Java EE platform focused on:

- dependency injection and
- lifecycle management of stateful objects bound to contexts

It's pretty easy to get started with CDI. You typically just 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 like WildFly, WebSphere, Apache Karaf, etc. Many containers support CDI by 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 a `CamelContext` for you. It searches for any routes as well and adds them to the context. Common Camel services like Endpoints, ProducerTemplates, TypeConverters, and even the `CamelContext` itself are injectable via CDI annotations.

We first covered CDI in section 7.2.2 so many of the basic concepts are discussed there. Also testing CDI applications was discussed in section 9.2.5. If you are unfamiliar with CDI, you should review these sections first. In this section we'll be focused on deployment to the WildFly and Apache Karaf containers.

15.6.1 WildFly deployment

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 we have:

```
<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 2 are probably obvious if you've read the previous sections that covered CDI. The last one is for an annotation specifically for WildFly-Camel. While not absolutely necessary, it is helpful to import the WildFly-Camel BOM so that you don't need to specify any dependency versions. You can do so like:

```
<dependencyManagement>
    <dependencies>
        <!-- WildFly Camel -->
        <dependency>
            <groupId>org.wildfly.camel</groupId>
            <artifactId>wildfly-camel</artifactId>
            <version>${version.wildfly.camel}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.wildfly.camel</groupId>
            <artifactId>wildfly-camel-patch</artifactId>
            <version>${version.wildfly.camel}</version>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Moving on to our route, we don't actually 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 4.4.0), you need either a `org.wildfly.extension.camel.CamelAware` or `org.apache.camel.cdi.ContextName` annotation. The route is shown below:

```
import org.apache.camel.builder.RouteBuilder;
import camelinaction.inventory.UpdateInventoryInput;
import org.wildfly.extension.camel.CamelAware;

@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");
    }
}
```

So 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 like:

```
<bean id="inventoryService" class="camelinaction.InventoryService"/>
```

But here, we have no such definition. With CDI you can create named beans via annotations. Let's look at our `InventoryService` bean in action:

```
import java.util.Random;
import javax.inject.Named;
import camelinaction.inventory.UpdateInventoryInput;
import camelinaction.inventory.UpdateInventoryOutput;

/**
 * Various service methods using in this example
 *
 * @version $Revision$
 */
@Named("inventoryService")
public class InventoryService {

    private Random ran = new Random();

    /**
     * To convert from model to CSV in a simply way
     */
}
```

```

public String xmlToCsv(UpdateInventoryInput input) {
    return input.getSupplierId() + "," + input.getPartId() + "," + input.getName() + ","
    + input.getAmount();
}

/**
 * To return an OK reply back to the webservice client
 */
public UpdateInventoryOutput replyOk() {
    UpdateInventoryOutput ok = new UpdateInventoryOutput();
    ok.setCode("OK");
    return ok;
}

/**
 * To simulate updating the inventory by calling some external system which takes a bit
 * of time
 */
public void updateInventory(UpdateInventoryInput input) throws Exception {
    // simulate updating using some CPU processing
    int sleep = ran.nextInt(1000);
    Thread.sleep(sleep);

    System.out.println("Inventory " + input.getPartId() + " updated");
}

}

```

Here you can see that we used the `javax.inject.Named` annotation to create a named bean instance we can refer to in our Camel route. The camel-cdi component starts up a `org.apache.camel.cdi.CdiCamelRegistry` to hold such beans in the `CamelContext`. To try out this example for yourself, change to the chapter15/cdi directory of the book's source and run:

```
mvn clean install -Pdeploy
```

This will deploy the example to a locally running instance of WildFly (if there is one running). One handy tip for deploying during development is to use the `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`:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.wildfly.plugins</groupId>
      <artifactId>wildfly-maven-plugin</artifactId>
      <version>1.1.0.Alpha11</version>
      <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 --&gt;
&lt;profiles&gt;
    &lt;profile&gt;
        &lt;id&gt;deploy&lt;/id&gt;
        &lt;properties&gt;
            &lt;deploy.skip&gt;false&lt;/deploy.skip&gt;
        &lt;/properties&gt;
    &lt;/profile&gt;
&lt;/profiles&gt;
</pre>

```

So what do you have to do if you instead decide to use CDI in an Apache Karaf container? Let's take a look next.

15.6.2 Karaf deployment

CDI-based Camel routes in Karaf are actually very similar to WildFly. The main differences are in the Maven pom.xml file. Let's look at the required dependencies first:

```

<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-cdi</artifactId>
</dependency>
<dependency>
    <groupId>javax.enterprise</groupId>
    <artifactId>cdi-api</artifactId>
    <version>1.2</version>
</dependency>

```

It's the same list as WildFly except that we now do not have an extra one for WildFly-Camel. There are a few more things to add in the pom.xml as well. Near the top of the file, we must specify that this is an OSGi bundle by using `<packaging>bundle</packaging>`. Also we must configure the maven-bundle-plugin to generate the bundle manifest correctly:

```

<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Bundle-SymbolicName>${project.groupId}.${project.artifactId}</Bundle-SymbolicName>

```

```

<Export-Package>camelinaction</Export-Package>
<Import-Package>*</Import-Package>
<Require-Capability>
    osgi.extender; filter:="(osgi.extender=pax.cdi)",
    org.ops4j.pax.cdi.extension; filter:="(extension=camel-cdi-extension)"
</Require-Capability>
</instructions>
</configuration>
</plugin>

```

While not completely necessary, its helpful to create a Karaf features file to make installing your Camel application easy for the user. Take this feature file for example:

```

<features>
    <repository>mvn:org.apache.camel.karaf/apache-camel/${camel-
        version}/xml/features</repository>

    <feature name='riderautoparts-cdi-karaf' version='${project.version}'>
        <feature version="${camel-version}">camel</feature>
        <feature version="${camel-version}">camel-cdi</feature>
        <feature>pax-cdi-weld</feature>
        <bundle start-level="90">mvn:com.camelinaction/riderautoparts-cdi-
            karaf/${project.version}</bundle>
    </feature>
</features>

```

Here all the user has to do is install the riderautoparts-cdi-karaf feature to get camel-cdi, the CDI implementation from the OPS4J PAX CDI project, and our Rider Auto application. We used the Weld CDI implementation here (see the pax-cdi-weld feature) but we could have also used the Apache OpenWebBeans CDI implementation by using the pax-cdi-openwebbeans feature.

Moving on to our route, we don't actually have to add any annotations to it for camel-cdi to pick it up. The route is shown below:

```

import org.apache.camel.builder.RouteBuilder;
import camelinaction.inventory.UpdateInventoryInput;

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");
    }
}

```

The inventory service here was added to the Camel registry with the `@Named("inventoryService")` annotation on the `InventoryService` class definition.

You can find the full example in the `chapter15/cdi-karaf` directory of the book's source.

15.7 Summary and best practices

In this chapter, we explored the internal details of how 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 important, 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.

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 are not.

In the second part of this chapter, we explored the art of deploying Camel applications as standalone Java applications, as web applications, in JEE 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, Camel can also be deployed in the cloud, or they can be started using Java Web Start.

Here are some pointers to help you out 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 on in the project that your application can be deployed and run in the environment.

In the next chapter, we'll take a tour of 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 using Jolokia
- Using the Camel management commands
- Overview of the Camel management API
- Accessing the Camel management API from within Camel and as well Java JMX clients
- Gathering details from the 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 also 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.

In this chapter, we'll review different 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 not only know the current values but also trends, such as whether the 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 different 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 and more mature and is often used in large system-management tools such as IBM Tivoli and HP OpenView. JMX, on the other hand, is a pure Java standard and is supported by some open source tools such as Jolokia, RHQ and Nagios.

In the following sections, we'll go over the three levels and look at some 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 only reveals whether Camel is running or not, but that will do for a basic check.

Suppose you have 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 easily can leverage. The scenario is 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 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 leverage 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 our own Camel route as a ping service we can also Jolokia. We can do this by using the Jolokia Java JVM agent which upon starting the Camel application will bootstrap Jolokia automatically.

Jolokia

Jolokia is an excellent library that makes JMX management fun again. As its core it is a 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 in your code. You can find more details about Jolokia at its website: <https://jolokia.org/>

You can try this 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/
```

... and Jolokia returns a JSON response with information about the Jolokia Agent running in the JVM as shown in figure 16.2.

```
{
  timestamp: 1440244859,
  status: 200,
  - request: {
      type: "version"
    },
  - value: {
      protocol: "7.2",
      - config: {
          maxDepth: "15",
          maxCollectionSize: "0",
          maxObjects: "0",
          discoveryEnabled: "true",
          agentContext: "/jolokia",
          historyMaxEntries: "10",
          agentId: "192.168.1.8-39186-12e6f711-jvm",
          agentType: "jvm",
          debug: "false",
          debugMaxEntries: "100"
        },
      agent: "1.3.1",
      - info: {
          product: "jetty",
          vendor: "Eclipse",
          version: "8.1.17.v20150415"
        }
    }
}
```

Figure 16.2 Web browser showing the returned information from calling the Jolokia Agent on the given url

Jolokia allows to read JMX attributes which we 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 we can see the uptime value is 8 minutes:

```
{"timestamp":1440247168,"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 CamelContext exposes in JMX, such as CamelVersion and ExchangesTotal.

We will 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.

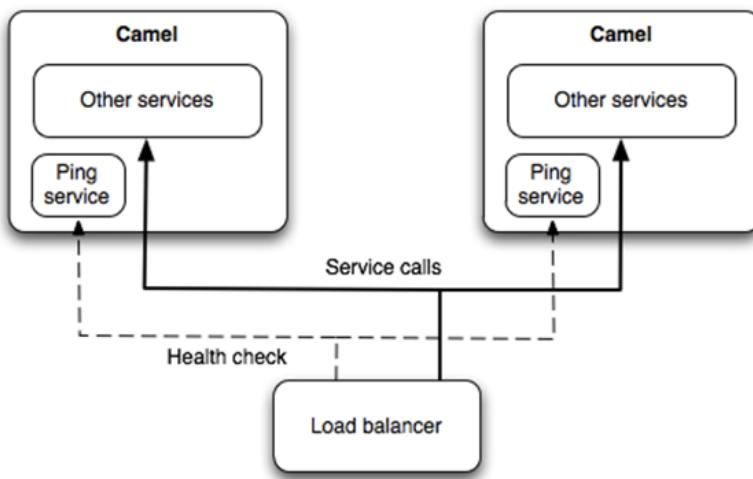


Figure 16.3 The load balancer uses health checks to ensure connectivity before it lets the service calls pass through.

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

The Simple Network Management Protocol (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 that issues notifications on low memory conditions. This allows you to use existing SNMP-aware tooling to monitor the JVM where Camel is running.

There is also a wide range of commercial and open source monitoring tools that use SNMP. Some are simpler and have a shell interface, and others have a powerful GUI. You may work in an organization that already uses a few selected 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 only exposing data at 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 you can 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 leverages the JMX technology. This is illustrated in figure 16.4.

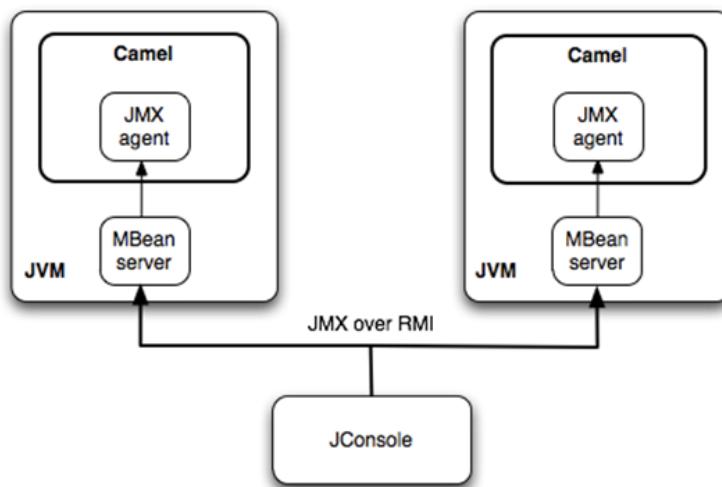


Figure 16.4 JConsole connects remotely to an MBean server inside the JVM, which opens up a world of in-depth information and management possibilities for Camel instances.

The JMX agent exposes remotely (over Remote Method Invocation) a wealth of standard details about the JVM, and some Camel information as well. 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.

So 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'll need to explicitly enable this in Camel.

We think this is important to cover thoroughly, so we've devoted 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 `INFO` level whether JMX is enabled or not:

```
2016-08-06 15:00:51,361 [viceMain.main()] INFO ManagedManagementStrategy      - JMX is
enabled
```

And when JMX is disabled:

```
2016-08-06 15:02:53,885 [viceMain.main()] INFO ManagedManagementStrategy      - JMX is
disabled
```

In this section we will look at two different management tools that can manage Java application using JMX. The first tool is JConsole which is shipped out of the box in Java. The other tool is using Jolokia and hawtio 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 can be used as well. However you would 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 should already list a process, and you can click the Connect button to connect JConsole to the Camel instance.

Figure 16.5 shows the Camel MBeans (Management Beans) that are visible from JConsole.

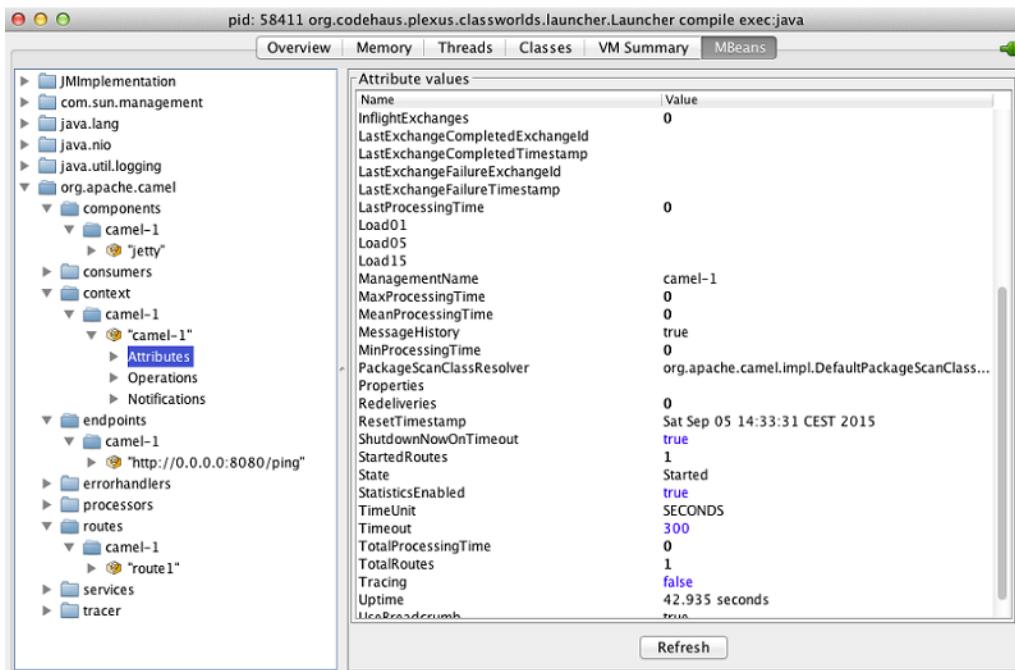


Figure 16.5 Camel registers numerous MBeans that expose internal details, such as usage statistics and management operations.

Camel registers many MBeans that expose statistics and operations for management. Those MBeans are divided into eleven 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 input 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 CamelContext itself. This is the MBean you need if you want to shut down Camel. |
| Dataformats | Lists the data formats in use. |
| Endpoints | Lists the endpoints in use. |

| | |
|---------------|---|
| Errorhandlers | 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. |
| Processors | Lists all the processors (Enterprise Integration Patterns) in use. The EIPs provides different runtime information and operations you can access. For example the Content Based Router EIP contains statistics how many times each predicate has matched. |
| Producers | Lists all the output producers for the Camel routes. Some producers have additional information and operations such as JMS, SEDA and File/FTP producers. |
| Routes | Lists all the routes in use. Here you can obtain route statistics, such as the number of messages completed, failed, and so on. |
| Services | Lists miscellaneous services in use. |
| Threadpools | 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. |
| Tracer | 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'll cover the use of the Tracer in detail in section 16.3.4. |

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 the `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 will look something like this:

```
2016-08-06 15:58:37,264 [viceMain.main()] INFO ManagedManagementStrategy - JMX is
enabled
2016-08-06 15:58:37,267 [viceMain.main()] INFO DefaultManagementAgent -
ManagementAgent detected JVM system properties:
```

```
{org.apache.camel.jmx.createRmiConnector=true}
2016-08-06 15:58:37,492 [99/jmxrmi/camel] INFO  DefaultManagementAgent      - JMX
Connector thread started and listening at:
service:jmx:rmi:///jndi/rmi://davsclaus.air: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 create an instance of `DefaultManagementAgent`, configure it to create a connector, and tell Camel to use it. The agent can also configure the registry port by using the `setRegistryPort` method.

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

        DefaultManagementAgent agent = new DefaultManagementAgent(context);
        agent.setCreateConnector(true);
        context.getManagementStrategy().setManagementAgent(agent);

        context.addRoutes(new PingService());
        context.start();
    }
}
```

But there is a simpler way: you can configure the settings directly using the `ManagementAgent` interface, as follows:

```
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>
```

The `<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. However it has its serious drawbacks as its a Java only technology that requires clients to use Java as well (or CORBA). Also as we have witnessed remote management requires using a connector port (usually 1099 as default) and then the JVM assigns a random port as well that are in use. This is seriously not firewall friendly, where organizations has to open ports in their network for JMX remote management. For more details see for example what Jolokia writes: <https://jolokia.org/features/firewall.html>

So how can clients using other technology access and manage Java applications, which is also firewall friendly? A great answer to this is to use Jolokia.

16.2.3 Using Jolokia to manage Camel

Jolokia allows to access JMX using HTTP that is an ubiquitous technology that can be accessed from a web browser, a HTTP client from the command line, or a web console such as hawtio.

TIP We recommend to read about the features of Jolokia from its website: <https://jolokia.org/features-nb.html>

Jolokia offers a number of different agents providing Jolokia services in various environments. In this chapter we will cover these three agents:

- WAR Agent for deployment as a web application in a 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 most easy to use so we 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 pre existing Jolokia WAR, or embed Jolokia into you existing WAR application. In this section we will 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 if Jolokia is running, by opening the following url: <http://localhost:8080/jolokia-war-1.3.5> (assuming we are using version 1.3.5 of Jolokia)

5. If Jolokia is running the webpage should output a Jolokia status page (similar to figure 16.2)

By installing the Jolokia WAR once into Apache Tomcat allows to using Jolokia to manage any of the applications that are deployed in Tomcat. So for example if you deploy 5 Camel applications and 3 other applications, then 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

Another easy way of using Jolokia is by embedding Jolokia to your existing WAR application. This can be done by adding a dependency to Jolokia in the Maven pom.xml file:

```
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
    <version>1.3.5</version>
</dependency>
```

And then add Jolokia to the web.xml file to expose the Jolokia as a 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 source code for this book contains an example in the chapter16/jolokia-embedded-war which 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 by

```
mvn clean install
```

And then copy the generated WAR file target/chapter16-jolokia-war.war to the deploy directory of Apache Tomcat.

If you are using Apache Tomcat with the default HTTP port 8080, then 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 Jolokia WAR Agent together with your Camel application in the same deployment unit is illustrated in figure 16.6.

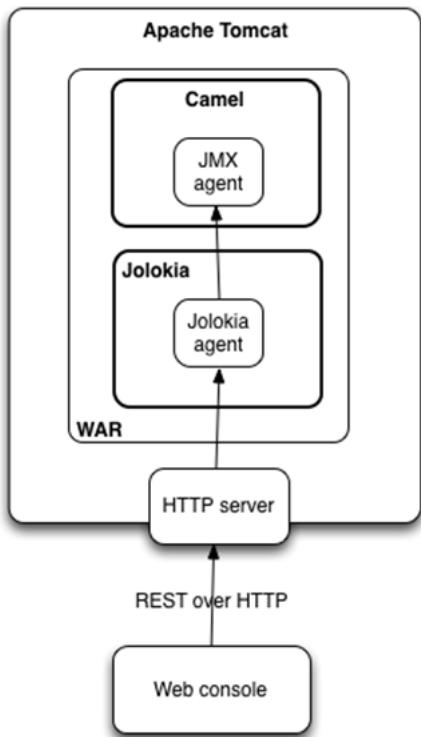


Figure 16.6 Jolokia is embedded together with Camel into the same WAR deployment unit which is running inside Apache Tomcat. The Jolokia service is exposed using the HTTP server of Apache Tomcat which the web console can access to obtain information about the running Camel application.

HAWTIO THE HOT WEB CONSOLE

As a cool web console you can use hawtio. For example try the example from [chapter16/jolokia-war-embedded](#) and deploy hawtio as well into Apache Tomcat. The hawtio console will auto detect the Camel applications running in the JVM which allows you to use the plugin to visualize the running Camel routes and as well manage those, and much more. hawtio is covered in much more details in the next chapter where we talk about Camel tools.

The next kind of Agent is the OSGi Agent which works in OSGi containers such as Apache Karaf and JBoss Fuse. As Jolokia is pre-installed in JBoss Fuse we will use Apache Karaf as demonstration how to install and use Jolokia.

USING JOLOKIA WITH APACHE KARAF

Using Jolokia on Apache Karaf is easy, as you can install Jolokia using the Karaf Shell. After Karaf has been started then type in the Shell (Karaf 3 onwards)

```
feature:install war
install -s mvn:org.jolokia/jolokia-osgi/1.3.5
```

Jolokia is then available on the default Karaf HTTP port (8181) using the context path `jolokia`. So you should be able to access Jolokia using the following url:

```
http://localhost:8181/jolokia/
```

Okay lets move on and talk about the last kind of Jolokia Agent which is the Java JVM Agent.

USING JOLOKIA JAVA AGENT

Jolokia provides a JVM agent (aka Java Agent) which does not need any special runtime environment, as it uses the embedded HTTP server from Java itself (requires Java 1.6 onwards). This agent is the most generic, and can be used for any Java application. It's particular useful when the other specialized agents does not fit. The principle how this agent works is shown in figure 16.7.

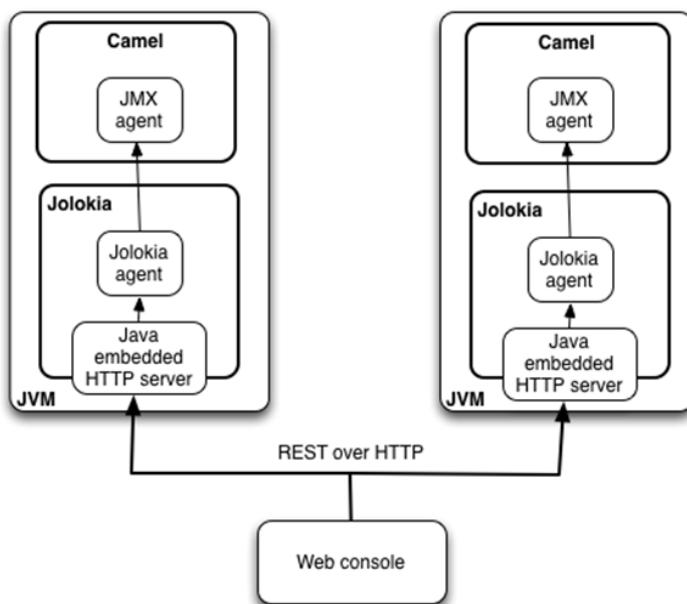


Figure 16.7 A web console (or HTTP client) connects remotely to a JVM of choice to manage using HTTP. The JVM embeds the HTTP server and the Jolokia Agent bridges HTTP to JMX so the web console can gather all JMX information from the Camel application.

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 we 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 source code for the book contains an example how to use the Jolokia Java Agent in the chapter16/jolokia directory, which you can try with the following Maven goal:

```
mvn compile exec:java
```

If you are not familiar with JVM agents then 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 we need to use Maven to download the needed dependencies, which can be done once:

```
mvn dependency:copy-dependencies
```

Then we need to build the example:

```
mvn clean install
```

And finally we can run the example using this rather long command:

```
java -javaagent:lib/jolokia-jvm-1.3.5-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 freaking hot and awesome

We highly recommend Jolokia, it is a awesome library that makes JMX fun and easy to use again. It is a really hot and awesome library, and a little known fact is the author of Jolokia, Roland Huss, is a avid chili fan, and therefore named the project after the most strong 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 all our software projects. For example as in JBoss Fuse, and all the Java based Docker images, comes with Jolokia JVM Agent installed.

Now that you've seen how to check the health of your applications, as well as dipping 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 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 there are errors 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 this by tracking the traces that various activities leave behind. By configuring Camel to leave traces, you can get a 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 leverage.

Let's look at how you can 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 a solution 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.

Centralized logging

If you have been following the DevOps movement you may be familiar with concepts such as data analytics, centralized logging, and data visualizations. A popular open source stack fits this bill known as the ELK stack. Elasticsearch for data analytics, Logstash (or Fluentd) for centralized logging, and Kibana for data visualization in a modern web console.

We will touch the topic about DevOps and the ELK stack in chapter 18, where we cover Camel in the cloud, where we cover the fabric8 project that offers the ELK stack out of the box.

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 leverage 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 `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 `DEBUG` logging level and an incredible amount at `TRACE` logging level. These levels are only appropriate for development, where the core logs provide great details for the developers.

In production, you’ll want to use `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’ll cover in section 16.3.4.

The core logs in production usage usually only provide 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 the situation where orders flowing in from CSV files are wire-tapped to an audit service before moving on for further processing.

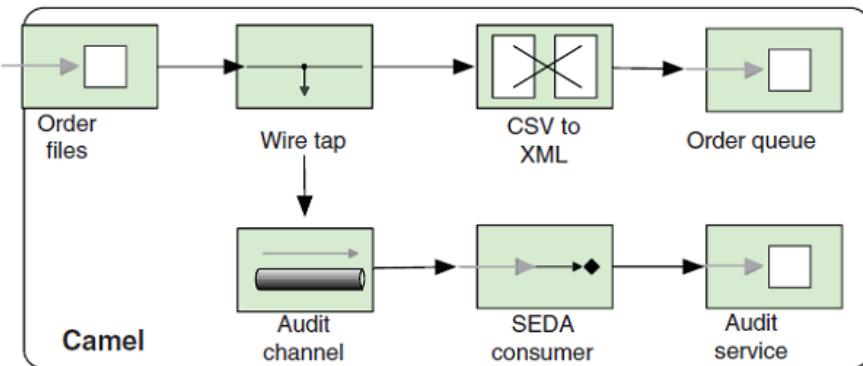


Figure 16.8 Using a wire tap to tap incoming files to an audit service before the file is translated to XML and sent to an order queue for further processing

Implementing the routes outlined in figure 16.8 in Camel is fairly straightforward as shown in listing 16.1:

Listing 16.1 - Using WireTap to tap messages for audit logging

```

public void configure() throws Exception {
    from("file://rider/orders")
        .wireTap("seda:audit") ①
        .bean(OrderCsvToXmlBean.class)
        .to("jms:queue:orders");

    from("seda:audit")
        .bean(AuditService.class, "auditFile"); ②
}

```

- ① Wiretap message to a seda queue
- ② Use a bean to implement logic to write to audit log

The first route is routing incoming order files. These are wire-tapped 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, which follows in listing 16.2:

Listing 16.2 - Implementation of a simple audit logging service using Java bean

```

public class AuditService {

    private Log LOG = LogFactory.getLog(AuditService.class); ①

    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);
    }
}

```

- ① Logger to use
 ② Logging an audit trail of the message

This implementation of the `AuditService` bean has been kept simple by logging the audit messages using a Java logger library ① . The Java bean constructs a logging message using various parts of the message body which then gets logged ② .

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 source code for the book contains this example in the chapter16/logging directory, which you can try 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 leverage the Log component, you simply 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:

```

2016-08-10 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-air-55845-1439210860847-0-1]]]
2016-08-10 14:47:42,351 [et/rider/orders] INFO asXml
- Exchange[ExchangePattern: InOnly, BodyType: String, Body:
<order><id>123</id><customerId>4444</customerId><date>20160810</date><item><id>222</id><amount>1</amount></item></order>]

```

By default, the Log component will log the message body and its type at `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 by 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 by using the following configuration:

```
log:incoming?showExchangePattern=false&showBodyType=false&maxChars=100
```

That results in the following output:

```
2016-08-10 14:50:36,575 [et/rider/orders] INFO  asXml
- Exchange[Body:
<order><id>123</id><customerId>4444</customerId><date>20160810</date><item><id>222</id><amount...>
```

TIP The Log component has a `showAll` option to log everything from the Exchange.

The Log component is used to log information from the 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. This is 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:

```
2016-08-10 14:52:32,576 [et/rider/orders] INFO route1
  - We got incoming file someorder.csv containing: 123,4444,20160810,222,1
```

The Log EIP will, by default, log at `INFO` level using the route ID as the logger name. In this example, the route was not explicitly named, so Camel assigned it the name `route1`.

Using the Log EIP from XML DSL is also easy, as shown here:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://target/rider/orders"/>
    <log message="Incoming file ${file:name} containing: ${body}"/>
    <bean beanType="camelinaction.OrderCsvToXmlBean"/>
    <to uri="jms:queue:orders"/>
  </route>
</camelContext>
```

The XML example is also provided in the source code for the book, which you can try 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 below 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 this by assigning a unique ID to each created message. In Camel, this ID is the `ExchangeId`, which you can grab from the Exchange using the `exchange.getExchangeId()` method.

TIP You can tell the Log component to log the ExchangeId by 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.

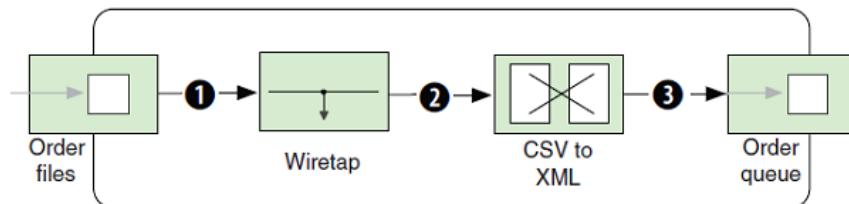


Figure 16.9 Tracer sits between each node in the route (at 1, 2, and 3) and traces the message flow.

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. The Channel has multiple purposes, such as error handling, security, and interception. Because the Tracer is implemented as an interceptor, it falls under the control of the Channel, which at runtime will invoke it.

To use the Tracer, you need to enable it, which is easily done in either the Java DSL or XML DSL. In the Java DSL, you 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 the Tracer from `<camelContext>` as follows:

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

When running with Tracer enabled, Camel will record trace logs at `INFO` level, which at first may seem a bit verbose. To reduce the verbosity, we have configured the Tracer to not show properties and headers. Here is an example of Tracer output:

```
2016-08-13 15:37:04,072 [et/rider/orders] INFO Tracer
- ID-davsclaus-air-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
```

```

    file based: GenericFile[ID-davsclaus-air-56347-1439472468659-0-117]]
2016-08-13 15:37:06,076 [et/rider/orders] INFO Tracer
- ID-davsclaus-air-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
  file based: GenericFile[ID-davsclaus-air-56347-1439472468659-0-117]]
2016-08-13 15:37:06,076 [- - seda://audit] INFO Tracer
- ID-davsclaus-air-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-air-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, there are two different IDs (highlighted in bold) in play: `ID-davsclaus-air-56412-1439473020118-0-202` and `ID-davsclaus-air-56412-1439473020118-0-205`. You may wonder why we have two IDs when there is 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, the Tracer outputs which route the message is currently at, 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 the 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.

CUSTOMIZING THE TRACER

We just said that we had customized the Tracer to be less verbose. This can be done by defining a bean in the Registry with the bean ID `traceFormatter`. In XML DSL this is easy—all you do is this:

```

<bean id="traceFormatter"
      class="org.apache.camel.processor.interceptor.DefaultTraceFormatter">
  <property name="showProperties" value="false"/>
  <property name="showHeaders" value="false"/>
</bean>

```

The formatter has many other options that you can read about in its online documentation (<http://camel.apache.org/tracer>). One of the options is `maxChars`, which can limit the message body being logged. For example, setting it to a value of 200 will limit the Tracer to only output at most 200 characters.

To customize the Trace from Java DSL is little bit different than XML DSL. In the `configure` method of the `RouteBuilder` class, you can do as follows:

```

public void configure() throws Exception {
    context.setTracing(true); 1
    Tracer tracer = (Tracer) context.getDefaultTracer(); 2
    tracer.getDefaultTraceFormatter().setShowProperties(false);
}

```

```
tracer.getDefaultTraceFormatter().setShowHeaders(false);
```

The tracer needs to be enabled ❶ and then customized in the following. To obtain the tracer you use the `getDefaultTracer` ❷ method from `CamelContext`. From the `Tracer` you can then access the `DefaultTraceFormatter` where you can use the setters to configure the options.

If you have many routes, the Tracer will output a lot of logs. Fortunately, you can customize the Tracer to only trace certain routes. You can even do this at runtime using JMX, but we'll get to that a bit later.

ENABLING OR DISABLING TRACER FOR SPECIFIC ROUTES

Remember that you can enable the Tracer from the `<camelContext>` tag? You can do the same with the `<route>` tag.

Suppose you only wanted to trace the first route—you could enable the Tracer on that particular route, as shown here:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route trace="true">
        <from uri="file:///rider/orders"/>
        <wireTap uri="seda:audit"/>
        <bean beanType="camelaction.OrderCsvToXmlBean"/>
        <to uri="jms:queue:orders"/>
    </route>
    <route>
        <from uri="seda:audit"/>
        <bean beanType="camelaction.AuditService"/>
    </route>
</camelContext>
```

Doing this from the Java DSL is a bit different. You need to do it using the fluent builder syntax by using either `tracing()` or `noTracing()`, as follows:

```
public void configure() throws Exception {
    from("file:///target/rider/orders")
        .tracing()
        .wireTap("seda:audit")
        .bean(OrderCsvToXmlBean.class)
        .to("jms:queue:orders");

    from("seda:audit")
        .bean(AuditService.class, "auditFile");
}
```

If a route isn't explicitly configured with a Tracer, it will fall back and leverage the configuration from the `CamelContext`. This allows you to quickly turn tracing on and off from the `CamelContext` but still have some special routes with their own settings.

The Tracer can also be managed using JMX. This allows you to enable tracing for a while to see what happens and identify issues.

MANAGING TRACER USING JMX

The Tracer can be managed from the JMX console in two places. You can enable or disable tracing at either the context or at routes. For example, to enable tracing globally, you could change the tracing attribute to `true` at the CamelContext MBean. You could do the same on a per route basis using the Routes MBeans.

You can configure what the Tracer logs from the `Tracer` MBean, as shown in figure 16.10.

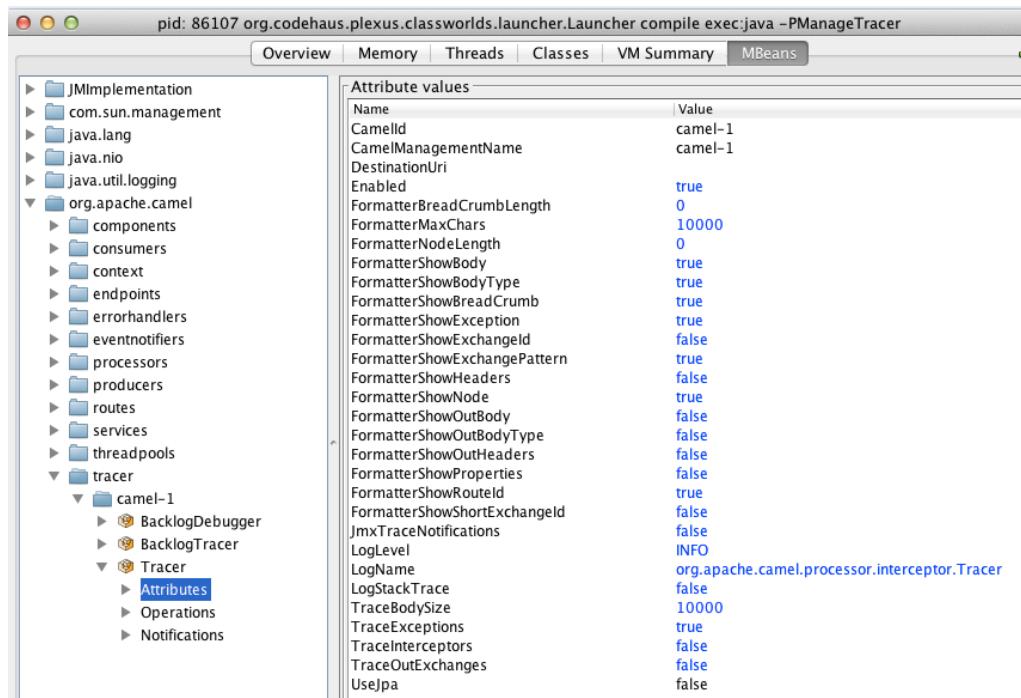


Figure 16.10 Managing the Tracer from JMX allows you to customize the trace logging and change many attributes at runtime.

We've prepared an example for you to try in the source code for the book. First, run this Maven goal in the chapter16/tracer directory:

```
mvn compile exec:java
```

This will start an application that will run for a while.

Then start `jconsole` and connect to the application. Click on the MBeans tab and expand the `org.apache.camel` node in the tree. Figure 16.11 shows where you're going.

| | |
|---------------------------------|----------------------------------|
| LastExchangeCompletedExchangeId | ID-davsclaus-air-56489-143947 |
| LastExchangeCompletedTimestamp | Thu Aug 13 15:56:18 CEST 2015 |
| LastExchangeFailureExchangeId | |
| LastExchangeFailureTimestamp | |
| LastProcessingTime | 6005 |
| Load01 | |
| Load05 | |
| Load15 | |
| ManagementName | camel-1 |
| MaxProcessingTime | 6018 |
| MeanProcessingTime | 4004 |
| MessageHistory | true |
| MinProcessingTime | 2000 |
| PackageScanClassResolver | org.apache.camel.impl.DefaultPac |
| Properties | |
| Redeliveries | 0 |
| ResetTimestamp | Thu Aug 13 15:54:17 CEST 2015 |
| ShutdownNowOnTimeout | true |
| StartedRoutes | 2 |
| State | Started |
| StatisticsEnabled | true |
| TimeUnit | SECONDS |
| Timeout | 300 |
| TotalProcessingTime | 160167 |
| TotalRoutes | 2 |
| Tracing | true |
| Uptime | 2 minutes |
| UseBreadcrumb | true |
| UseMDCLogging | false |

Figure 16.11 To enable tracing, select the `CamelContext` under the `Context` node and change the `Tracing` attribute from `false` to `true`.

Click on the value for the `Tracing` attribute, which should be editable. Change the value from `false` to `true`, and press Enter to confirm. You should be able to see the changes in the console logged by the application.

Spend some time playing with this. For example, change some of the other options on the Tracer.

Distributed tracing

What we are covering in this chapter is tracing Camel applications in a non-clustered environment. In a distributed system you need something more powerful to help orchestrate and gather data from all the running services in the cluster. In the following chapter we will talk about zipkin which is used for distributed tracing.

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 very big, it may feel like you're looking for a needle in a haystack. Sometimes you might prefer to have Camel call you when particular events occur. This is 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'll 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 Apache Commons Logging framework. This means you can use 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 leverage 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 notifications, you can be overloaded with thousands of notifications. That's why you should always filter out unwanted notifications. The `PublishEventNotifier` will leverage 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 wish 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().setEventNotifier(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 wish to use. In this case, you're only interested in 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 16.12 illustrates the situation.

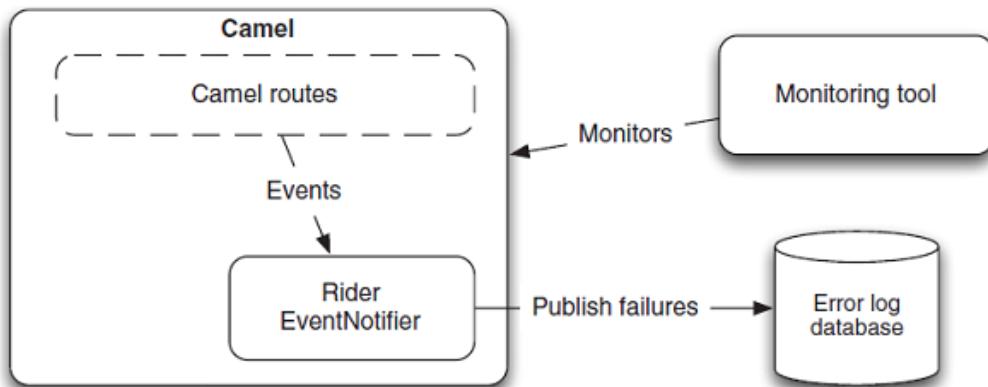


Figure 16.12 Failure events must be published into the centralized error log database using the custom `RiderEventNotifier`.

They decide to implement a custom event notifier named `RiderEventNotifier`, which uses their own Java code, allowing ultimate flexibility. The following listing shows the important snippets of how to implement this..

In listing 16.3, you extend the `EventNotifierSupport` class, which is 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
    }

    public void notify(EventObject eventObject) throws Exception {   ②
        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
        }
    }

    protected void doStart() throws Exception {}
    protected void doStop() throws Exception {}
}

```

① Filter which events to trigger upon

② Accepted events

③ Publishes failure events

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 ① in this example. If the `isEnabled` method returns `true`, the event is propagated to the `notify` method ②. 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 #e.

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 source code for the book 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, we'll take a further look at how to manage both Camel and your custom Camel components.

16.4 Managing Camel applications

We already touched on how to manage Camel in section 16.2, where we covered how to use JMX with Camel. In this section, we'll take deep dives into various management-related topics:

- *Camel application lifecycle* - How to control your Camel application, such as stopping and starting routes, and much more using a broad range of ways with JMX, Jolokia, hawtio, Camel commands, and the controlbus component
- *Camel management API* - Learn about the programming API from Camel that defines the management API. How end users can program using this API with standard JMX code, or with APIs from Camel.
- *Performance statistics* - Discover what key metrics Camel capture about your Camel application performance, and how you can access these metrics for custom reporting and hook into monitoring and alert tools.
- *Management enabling custom components* - Learn how you can program your custom Camel components and Java beans, so they are management enabled out of the box, as if they were first class from the Camel release.

We'll start by looking at how you can manage the lifecycles of your Camel applications.

16.4.1 Managing Camel application lifecycles

It's essential to be able to manage the lifecycles of your Camel applications. 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 this, you connect to the application with JMX as you learned in section 16.2. Figure 16.13 shows JConsole with the route in question selected.

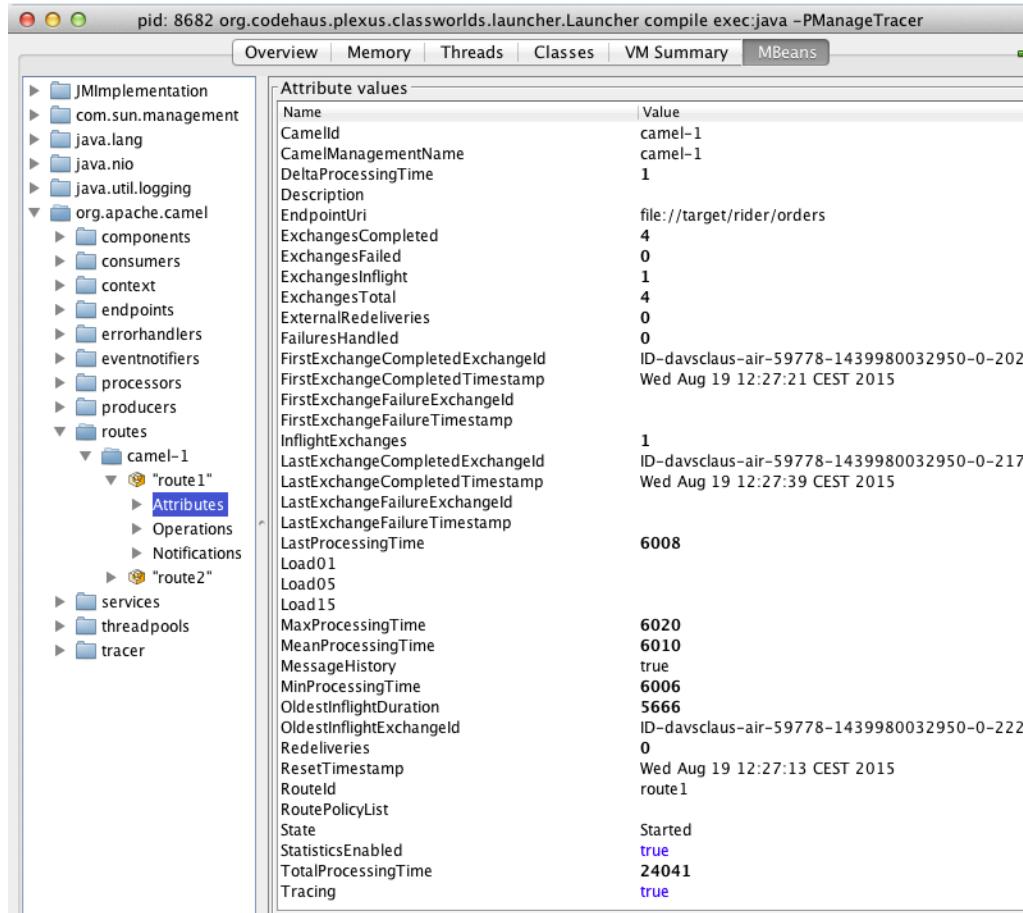


Figure 16.13 Selecting the route to manage in JConsole

As you can see in figure 16.12, `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, suspend 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`.

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 to invoke JMX operations which we can use to start and stop Camel routes. We will use an example from the source code to demonstrate this. In the chapter16/jolokia-embedded-war run the following Maven goal:

```
mvn clean install
```

And then copy the WAR file to a running Apache Tomcat in the webapps directory, such as:

```
cp target/chapter16-jolokia-embedded-war.war /opt/apache-tomcat-8.0.9/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 its stopping the route.

```
2016-09-19 11:05:20,700 [0 - timer://foo] INFO  route1
  - I am running.
2016-09-19 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. So how did we know this command? We used hawtio which can display the Jolokia REST call for each JMX attribute or operation.

Now download hawtio and install it into the running Apache Tomcat, such as by copying the WAR file

```
cp ~/Downloads/hawtio-default-1.4.68.war /opt/apache-tomcat-8.0.9/webapps/hawtio.war
```

And then access hawtio from a webbrowser: <http://localhost:8080/hawtio/>

The JMX tab in hawtio is similar to JConsole with a JMX tree on the left hand 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. Then find the stop operation and click it. This should lead to where figure 16.14 shows:

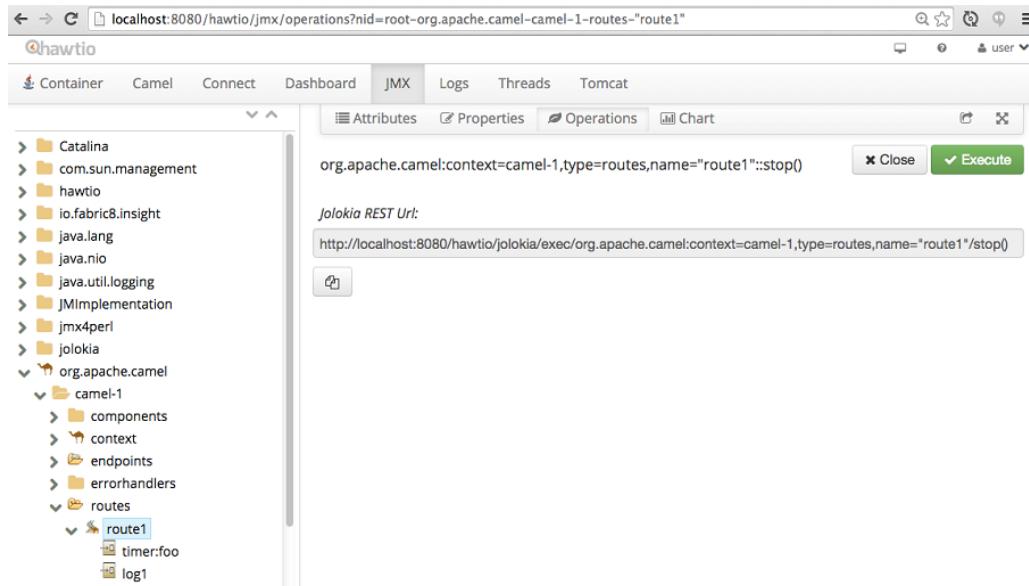


Figure 16.14 Using hawtio to manage a Camel application. In the JMX tab the Camel route mbean is selected in the tree, and the stop operation is selected which allows us to invoke the operation using the Execute button. The corresponding Jolokia REST url is also shown.

As figure 16.14 shows you can find every Jolokia REST url using the hawtio web console.

TIP To manage Camel routes with hawtio you will find better use of the Camel tab instead of the JMX tab. In the Camel tab there is display to list all the routes in a table where you can easily select one or more routes to start or stop with a click of a button.

Apache Camel also comes with a set of Camel commands which can be used for management and to access information about the Camel application in a similar fashion as the hawtio web console can do.

16.4.3 Using Camel commands to manage Camel

The Camel commands are intended for managing Camel in Apache Karaf. To run Camel on Karaf you need to start Apache Karaf from a shell:

```
bin/karaf
```

And wait for Karaf to startup and be ready. Then install Camel in Karaf:

```
feature:repo-add camel 2.19.0
feature:install camel
```

... where 2.19.0 is the Camel version to install.

Now we need to install a Camel application in Karaf, which can be done using the chapter16/jolokia-karaf example from the source code of the book. At first we need to build this example:

```
mvn clean install
```

And then from the Karaf command line type:

```
install -s mvn:camelinaction/chapter16-jolokia-karaf/2.0.0
```

.. which will install and start the example (the `-s` flag refers to start).

The Camel command is accessible from the Karaf shell, by typing `camel` and then hit the TAB key (Karaf shell has TAB completion), which then list all the Camel commands.

For example to show all the Camel applications type

```
camel:context-list
```

.. which in our example will only list one Camel application

```
karaf@root()> camel:context-list
Context      Status      Uptime
-----
camel-1     Started    3 minutes
```

The command `route-list` lists all the routes, so we can use that to know the id's of the routes if we want to manage those, such as to starting and stopping them as shown how can be done in the following:

```
karaf@root()> camel:route-list
Context      Route      Status
-----
camel-1     route1    Started
karaf@root()> camel:route-stop route1
karaf@root()> camel:route-list
Context      Route      Status
-----
camel-1     route1    Stopped
```

Notice how the status of the route changed from `Started` to `Stopped`. Likewise we can start the route again with the `route-start` command.

Stopping vs suspending a Camel route

Stopping a route will stop any resources the route consumer has started such as connection listeners. In case you want to temporary stop the route for a period of time, then consumer suspending the route instead. This operation is lighter as it attempts to not intake new messages, but keep the resources alive.

TIP To learn more about a command in Karaf you can type `help command-name`, for example to learn about the Camel route - show command type: `help camel:route-show`.

Installing hawtio in Apache Karaf

You can easily install hawtio as a web console in Karaf in a similar way we installed Camel.

```
feature:repo-add hawtio 1.4.68
feature:install hawtio
```

And then you can access the hawtio web console from a web browser using the following url: localhost:8181/hawtio
And to login to Karaf use karaf/karaf as the default username and password.

The Control Bus EIP pattern from the EIP book is the next topic.

16.4.4 Using controlbus to manage Camel

So far the various ways we have looked at for managing and controlling your Camel applications has been from the *outside*, but what if you want to do this from the *inside*? This is what the Control Bus EIP do, 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.

Lets illustrate how this works with a little example as shown in listing 16.4.

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
                .route().routeId("ping")
                    .transform(constant("PONG\n"))
            .endRest()

            .get("route/{action}")
                .toD("controlbus:route?routeId=ping&action=${header.action}"); ③
    }
}
```

① Using restlet component on port 8080 as rest server

② Ping service as rest

③ Control bus to control the route

When using the Camel rest-dsl (we covered rest-dsl in chapter 10) we need to configure which Camel component to use as the restlet server, in this example we use the restlet component ①. The ping service is a HTTP GET operation that return the pong response ②. The following rest service is using the context-path `route/{action}` where action is dynamic value that are bound as a Camel header, and therefore we are using dynamic-to as the controlbus endpoint. This allows us to specify the action parameter using the dynamic value of the header ③.

The source code of the book 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 below:

```
$ 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
```

In the example above we are using three of the possible action options the controlbus accepts. All the possible values the action option supports is listed in table 16.2.

Table 16.1 Supported values of the action option from the controlbus component

| Value | Description |
|---------|---|
| start | To start the route. There is no return value. |
| stop | To stop the route. There is no return value. |
| suspend | To suspend the route. There is no return value. |
| resume | To resume the route. There is no return value. |
| status | To get the route status returned as plain text value such as: Started, Stopped. |
| stats | To get the route performance status returned in XML format. |

The controlbus executes the action synchronously by default. For example if stopping a route takes 15 seconds to stop graceful, then that action has to complete before the Camel routing engine can continue routing the message. If you want to execute the task asynchronously then this can be done by setting the option `async` to `true`, as shown:

```
.toD("controlbus:route?routeId=ping&action=${header.action}&async=true")
```

We have so far seen how to manage Camel applications using various tools such as JMX, Camel commands and the controlbus component. They all operate on top of the Camel management API which we will take a 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 by using annotations to declare whether its a read-only or read-write attribute, or an operation. For example the `ManagedCamelContextMBean` is the management API of the `CamelContext`. Code listing 16.5 shows snippets of the source code for this interface.

Listing 16.5 - Source code of the 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
    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;
}
```

- ① Expose attribute for management (read-only on getter)
- ② Expose operation for management

In the `ManagedCamelContextMBean` interface each getter/setter becomes a JMX attribute by annotation the getter and setter methods with the `@ManagedAttribute` annotation ① . The attribute can be read-only if there is only an annotation on the getter method. The attribute is read-write when both the getter and setter has been annotated. Operations is regular Java methods that may return a response or not (`void`). The operation ② in listing 16.5 does not 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 was listed in table 16.1. For example there is MBeans for all the Camel components, data formats, endpoints, routes, processors (EIPs), thread pools, and miscellaneous services. Each of these MBean are defined as interfaces in the `org.apache.camel.api.management.mbean` package.

In section 16.5.2 we will see an example how you can use 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 will 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 integrate Camel management from an existing Java management library or tool.

NOTE Clients using JMX to manage Java applications have some drawbacks such as the client must use Java and the network transport is using Java RMI, which is not firewall friendly, as multiple ports must be open. See the sidebar titled [JMX the good, bad and ugly](#) from section 16.2.2 for more details.

In this section we will 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 some information about the Camel application. This scenario is illustrated in figure 16.15.



Figure 16.15 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 JMX proxy.

USING STANDARD JAVA JMX API

For a Java JMX client to be able to remote manage another Java application, its required for the remote application to expose a JMX connector, the client can connect and use. The Camel application does this by setting the `createConnector` option to true as shown below:

```
context.getManagementStrategy().getManagementAgent().setCreateConnector(true);
```

When Camel starts, a JMX connector is created and listening on port 1099 (the default port number). The url clients need to use for connecting to Camel is logged:

```
JMX Connector thread started and listening at:  
service:jmx:rmi:///jndi/rmi://davsclaus.air:1099/jmxrmi/camel
```

This can be handy as the url is not easy to remember.

Listing 16.6 lists how you could write a Java application that connects to the remote Camel application and output 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

    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

        System.out.println("Version: " + client.getCamelVersion()); ③
        System.out.println("Uptime: " + client.getCamelUptime()); ③

        client.disconnect(); ④
    }
}
```

- ① JMX Url to connect to remote Camel application
- ② Connecting to the Camel application
- ③ Getting information about the Camel application
- ④ Disconnecting from the remote Camel application

There is more to this client as we have implemented the lower level JMX code in another class called `JmxCamelClient` which is listed in listing 16.7.

Listing 16.7 - Lower level JMX API to remote manage a Camel application

```
import javax.management.MBeanServerConnection;
import javax.management.ObjectName; ①
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class JmxCamelClient {
    private JMXConnector connector;
    private MBeanServerConnection connection;

    public void connect(String serviceUrl) throws Exception {
        JMXServiceURL url = new JMXServiceURL(serviceUrl); ②
        connector = JMXConnectorFactory.connect(url, null); ③
        connection = connector.getMBeanServerConnection(); ④
    }

    public String getCamelVersion() throws Exception {
        ObjectName on = new ObjectName("org.apache.camel:context=camel-1,type=context,name=\"camel-1\""); ⑤
        return (String) connection.getAttribute(on, "CamelVersion"); ⑥
    }
}
```

```

    }

    public String getCamelUptime() throws Exception {
        ObjectName on = new ObjectName("org.apache.camel:context=camel-
1,type=context,name=\"camel-1\"");
        return (String) connection.getAttribute(on, "Uptime");
    }

    public void disconnect() throws Exception {
        connector.close();
    }

}

```

- ① Only import standard Java JMX API
- ② Setup URL to connect to the remote JVM
- ③ Connect to the remote JVM using the URL
- ④ Open a connection the client uses
- ⑤ JMX ObjectName to get the version of Camel
- ⑥ Read the JMX attribute using the remote connection
- ⑦ Disconnect from the remote JVM

We can see from the top of listing 16.7 that this class only imports standard Java JMX API ① . To connect to a remote JVM three lines of code is 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 shutdown.

The lower level JMX code in listing 16.7 can also use the Camel management API.

USING JMX PROXY

When using 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.

Listing 16.8 shows how we could use these interfaces to simplify the code.

Listing 16.8 - Lower level JMX API using JMX Proxy to facade 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

public class JmxCamelClient2 {

    private JMXConnector connector;
    private MBeanServerConnection connection;
    private ManagedCamelContextMBean proxy;
}

```

```

public void connect(String serviceUrl) throws Exception {
    JMXServiceURL url = new JMXServiceURL(serviceUrl);
    connector = JMXConnectorFactory.connect(url, null);
    connection = connector.getMBeanServerConnection(); ②

    // create a 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\""); ③
    proxy = JMX.newMBeanProxy(connection, on,
        ManagedCamelContextMBean.class); ④
}

public void disconnect() throws Exception {
    connector.close();
} ⑤

public String getCamelVersion() throws Exception {
    return proxy.getCamelVersion(); ⑤
}

public String getCamelUptime() throws Exception {
    return proxy.getUptime(); ⑤
}
}

```

- ① Importing the Camel management API
- ② Connecting to the remote JVM
- ③ JMX ObjectName for the CamelContext
- ④ Creating a JMX Proxy using ManagedCamelContextMBean as facade
- ⑤ Read the JMX attributes using the facade

A difference between listing 16.7 and 16.8 is that we are now using the Camel management API as the facade ① . When doing this it is required to add camel-core on the classpath. The benefit of using the facade is the entire Camel management API is accessible, as if you are 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 listing 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 is using type-safe Java method calls.

The source code of the book contains this example in the chapter16/jmx-client directory. To run the example you need to start the Camel application first using:

```
mvn compile exec:java -Pserver
```

And then you can run the clients using:

```
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 only handle at most 5 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 have then been told to build a solution that can report to the central monitoring system how many orders are currently being throttled by Camel. How can you build such a solution?

With all the knowledge from this chapter you are in safe hands. You have learned that Camel exposes a wealth of information at runtime. Table 16.1 listed all the categorizes of the 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 discovers 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 would need to use the Camel management API as a client from within Camel. This can be done as we learned from the previous section 16.5.1 using the JMX and JMX Proxy technique. But hey you are using Camel so there is usually an easier way.

The easy way with Camel is using `CamelContext` go quickly get an MBean that represent a given EIP from any of the Camel routes. This can be done using the `getManagedProcessor` method from `CamelContext`.

Listing 16.9 shows how you can 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
    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);
        return throttler.getThrottledCount();      ②
    }                                              ③
}

```

- ① Class is CamelContextAware to have CamelContext injected
- ② Get the throttler MBean
- ③ Return the number of current throttled messages

The source code in listing 16.9 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 the `CamelContext` to lookup 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, maximum processing time, and much more. This information is available on three levels:

- *CamelContext* - Aggregated statistics for all the Camel routes and processors
- *Route* - Aggregate statistics for the current route and its processors
- *Processor* - Statistics for each individual processor

What information is exposed is defined in the interface `org.apache.camel.api.management.mbean.ManagedPerformanceCounterMBean` as JMX attributes. The follow 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;

@ManagedAttribute(description = "Delta Processing Time [milliseconds]")
long getDeltaProcessingTime() throws Exception;

```

All this information is available out of the box. For example figure 16.11 is a screenshot from JConsole with the CamelContextMBean. On this screenshot we can see some of the performance statistics such as the min/mean/max values.

An information that was recently added to Camel was what is the oldest duration of all the current inflight messages - in other words which of all the inflight messages is currently taking the most time.

```

@ManagedAttribute(description = "Oldest inflight exchange duration")
Long getOldestInflightDuration();

```

The source code for the book contains a little example how you can obtain this information every second and log to the console. This example is in the chapter 16/jmx-camel directory which you can run using the following Maven goal:

```
mvn test -Dtest=OldestTest
```

The implement is similar to what you have done in listing 16.9. The ManagedRouteMBean is retrieved from CamelContext using the following code:

```
ManagedRouteMBean route = context.getManagedRoute("myRoute",
    ManagedRouteMBean.class);
```

... where `myRoute` is the id of the route.

TIP The performance statistics can be dumped as XML using the `dumpStatsAsXml(boolean)` method. If the boolean parameter is `true` then 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 by using the camel-metrics component.

USING CAMEL-METRICS FOR ROUTE PERFORMANCE STATISTICS

The camel-metrics component allows you to easily capture route performance statics 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 an `org.apache.camel.spi.RoutePolicyFactory` which will create a new `RoutePolicy` for all your routes. The camel-metrics component offers the `MetricsRoutePolicyFactory` which can be used in Java or XML DSL as shown:

```
context.addRoutePolicyFactory(new MetricsRoutePolicyFactory());
```

And in XML DSL you just declare the factory as a `<bean>` and its automatic enabled:

```
<bean id="metricsFactory"
      class="org.apache.camel.component.metrics.routepolicy.MetricsRoutePolicyFactory"/>
```

This will use Dropwizard metrics to capture timings of how long time the route take 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 16.10 Accessing Dropwizard performance statics from Java code

```
MetricsRegistryService registryService = context.hasService(MetricsRegistryService.class);
①

if (registryService != null) {
    MetricRegistry registry = registryService.getMetricsRegistry(); ②

    long count = registry.timer("camel-1:foo.responses").getCount(); ③
    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("Foo metrics: count={}, mean={}, rate1={}, rate5={}, rate15={}", count, mean,
            rate1, rate5, rate15);
}
```

- ① Obtain the MetricsRegistryService from CamelContext
- ② Get hold of Dropwizard MetricRegisry that holds the statistics
- ③ Get the response timer for the foo route as count, mean, rate1, rate5 and rate15

The code from listing 16.10 are from an example from the book which is located in the chapter16/metrics directory.

To get hold of the Dropwizard metrics we need first to get hold of `MetricsRegistryService` from the `CamelContext` ①. The `MetricRegistry` ② then gives access to all the performance statistics which 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, and 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. This can be done using JMX which listing 16.11 shows how can be done:

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

MBeanServer server = context.getManagementStrategy().getManagementAgent().getMBeanServer(); ②

String json = (String) server.invoke(on, "dumpStatisticsAsJson", null, null); ③
```

- ① JMX ObjectName to the MetricsRegistryService
- ② Get hold of the MBeanServer
- ③ Invoke the dumpStatisticsAsJson operation

The code from listing 16.11 is from the same example as shown in listing 16.10. To get the performance statistics in JSON from JMX, we need to get hold of the JMX MBean `ObjectName` ① that has the `dumpStatisticsAsJson` operation ③.

Using JMX is in fact what hawtio leverages in the Camel plugin that can show the Dropwizard performance statistics in a graphical chart as shown in figure 16.16.

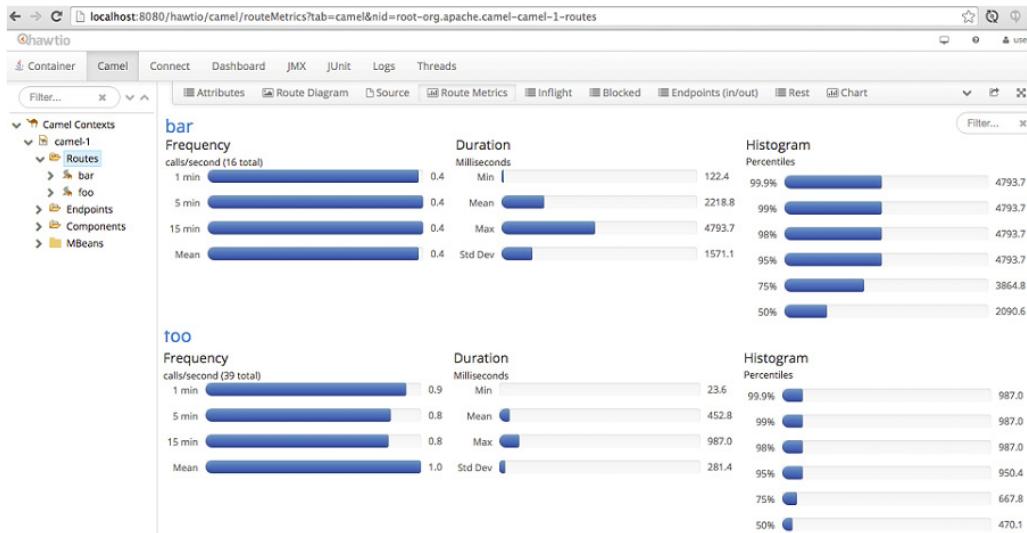


Figure 16.16

You can try this example by running the following goal from Maven:

```
mvn compile hawtio:run
```

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 would like to manage. This can also be done.

16.5.4 Management enable custom Camel components

Suppose Rider Auto Parts has developed a Camel ERP component to integrate with their 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 you can 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 source code for the book in the `chapter16/custom` directory.

Listing 16.12 Management enabling a custom endpoint

```
@ManagedResource(description = "Managed ERPEndpoint")
public class ERPEndpoint extends DefaultEndpoint {
```

```

private String name;
private boolean verbose;

public ERPEndpoint(String endpointUri, Component component) {
    super(endpointUri, component);
}

public Producer createProducer() throws Exception {
    return new ERPProducer(this);
}

public Consumer createConsumer(Processor processor) throws Exception {
    throw new UnsupportedOperationException("Consumer not supported");
}

public boolean isSingleton() {
    return true;
}

@ManagedAttribute(description = "Verbose logging enabled")
public boolean isVerbose() {                                ②
    return verbose;
}

@ManagedAttribute(description = "Verbose logging enabled")
public void setVerbose(boolean verbose) {                  ②
    this.verbose = verbose;
}

@ManagedAttribute(description = "Logical name of endpoint")
public String getName() {                                 ③
    return name;
}

public void setName(String name) {
    this.name = name;
}

@ManagedOperation(description = "Ping test of the ERP system")
public String ping() {                                    ④
    return "PONG";
}
}

```

- ① Expose class as MBean
- ② Expose attribute for management (read/write on getter and setter)
- ③ Expose attribute for management (read-only on getter)
- ④ Expose operation for management

If you've ever tried using the JMX API to expose the management capabilities of your custom beans, you'll know it's a painful API to leverage. It's better to go for the easy solution and leverage Camel JMX. You'll notice, in the source code from listing 16.6, 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 by using the

@ManagedAttribute ② annotation on the setter method. Likewise we can expose the name property as a managed read-only attribute when we 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 chapter16/custom directory to try out this example:

```
mvn compile exec:java
```

When you do, the console will output a log line every 5 seconds, as the route below 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 16.17 shows how this is done from JConsole.

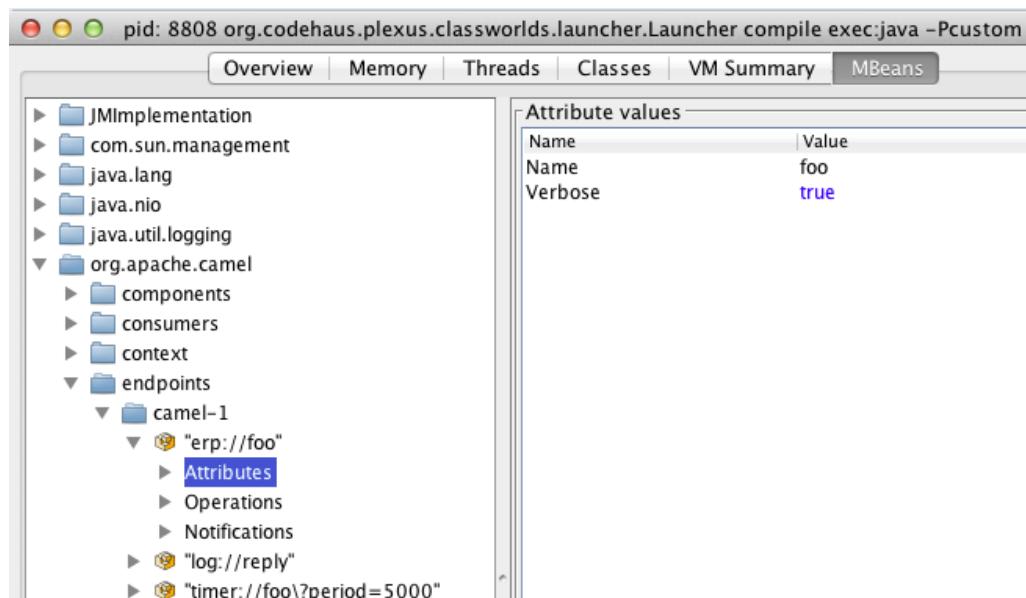


Figure 16.17 Enabling the Verbose attribute at runtime using JConsole. Click on Value column to edit and change the text from false to true as shown in the figure.

As you can see in figure 16.17, 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...`, as shown below:

```
Calling ERP with: Hello ERP calling at 12:53:12
2016-08-19 12:53:12,145 [0 - timer://foo] INFO reply
- Exchange[ExchangePattern: InOnly, BodyType: String, Body: Simulated response from
ERP]
Calling ERP with: Hello ERP calling at 12:53:17
2016-08-19 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
2016-08-19 12:53:22,145 [0 - timer://foo] INFO reply
- Exchange[ExchangePattern: InOnly, BodyType: String, Body: Simulated response from
ERP]
```

What you have just learned about management-enabling a custom component is in fact 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, allow you to manage the consumers to adjust how often they should trigger.

It is not only custom Camel components you can management enable using the Camel annotations. This is also possible to do on regular Java beans (POJOs).

16.5.5 Management enable custom Java beans

To management enable custom Java beans you use the same approach as for management enabling custom Camel components, by using the Camel management annotations.

Listing 16.13 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

```
import org.apache.camel.api.management.ManagedAttribute;
import org.apache.camel.api.management.ManagedOperation;
import org.apache.camel.api.management.ManagedResource;

@ManagedResource
public class HelloBean {①

    private String greeting = "Hello";

    @ManagedAttribute(description = "The greeting to use")②
    public String getGreeting() {
        return greeting;
    }

    @ManagedAttribute(description = "The greeting to use")②
    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }

    @ManagedOperation(description = "Say the greeting")③
    public String say() {
```

```

        return greeting;
    }
}

① Expose class as MBean
② Expose attribute for management (read/write on getter and setter)
③ Expose operation for management

```

By adding the Camel management annotations to the bean we can expose it as a MBean. This is done by adding the `@ManagedResource` on the class level ① . Attributes on the bean can be exposed using the `@ManagedAttribute` ② which supports read-only and read-write mode. In this example we have added the `@ManagedAttribute` on both the getter and setter, and hence the attribute is read-write. For read-only mode then the annotation should only be configured on the getter. A JMX operation can be exposed using the `@ManagedOperation` ③ which in this example will invoke 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 we learned from table 16.1.

The source code for the book 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")
    .beanRef("hello", "say")
    .log("${body}");
```

The application uses a Camel `Main` class to boot Camel. As part of the configuring of the main class, a `HelloBean` instance is created and enlisted in the Camel registry using the name `hello` as shown ① :

```
public static void main(String[] args) throws Exception {
    Main main = new Main();
    main.bind("hello", new HelloBean()); ①
    main.addRouteBuilder(new HelloRoute());
    main.enableHangupSupport();
    main.run();
}
```

- ① bind a `HelloBean` instance in the Camel registry

You can try this example by running the following Maven goal:

```
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 16.18.

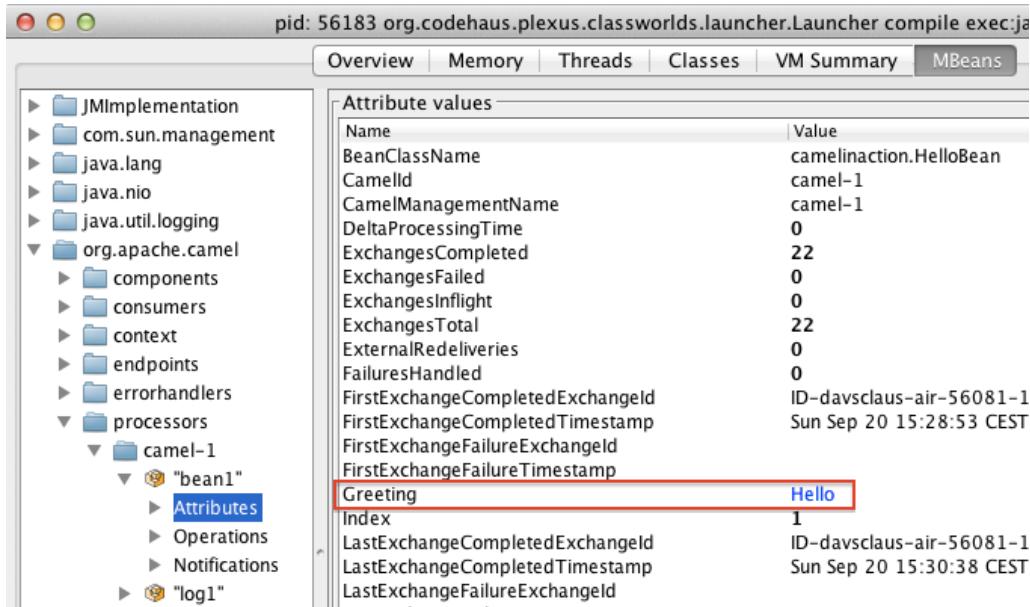


Figure 16.18 The HelloBean with the custom Greeting attribute is enlisted in the Camel processors tree, also inheriting the standard set of Camel JMX attributes such as all the performance details.

Congratulations! You have 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, we looked at how you can monitor your Camel applications using health-level checks. You learned that existing monitoring tools could be used via SNMP or JMX protocols. 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.

We also looked at what Camel has to offer in terms of logging. You learned about the Camel logs and how you can use custom logging. We also covered the Camel notification system, which is pluggable, allowing you to hook in your own notification adapter and send notifications to a third party.

Here are a few simple guidelines:

- *Involve the operations team.* Monitoring and management isn't an afterthought. 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 the Tracer to see how they're actually being routed. But beware; the Tracer can be very 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 management enable your custom Camel components and Java beans, and let clients easily access and management them with the help from Jolokia to use REST and HTTP.
- *Cloud, Containers, and DevOps.* Capturing key metrics and data analytics are in more demand. The rise of cloud platforms and the DevOps movement makes it easier for more companies to implement a solution. Take a look at chapter 18 for more details on using Camel in such platforms.

The next chapter is taking us from one Camel to a Camel herd, where we cover running many Camels using containers running in the cloud or a container platform such as Kubernetes. However before breaking into the exiting new world of containers we will talk about what clustering support there comes with various Camel components. You are know a very experience Camel jockey, its time to prove you can run ride in a Camel caravan.

19

Camel Tooling

This chapter 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
- Build custom tooling

This chapter talks you through some of the most powerful and known Camel tools that you can find on the internet.

We start with tooling that appeal to certain kind of people who may not yet know the true powers of Camel, and think that developing Camel application requires a graphical based end to end Camel editor. We then show you alternative tools you can use in your IDE editors that allows to do in-place code assistance to edit Camel endpoints in a more type-safe manner. We should you a Maven tool which is capable of scanning the source code and report any Camel configuration mistakes. You can also find tools to debug Camel applications by allowing you to step through the routes and inspect the Camel messages.

The last tool we demonstrate can inspect running Camel applications and visualize the Camel routes with real time metrics. And as well management functionality that allows to control your Camel applications such as starting and stopping routes.

You learn the secret of the Camel Catalog that 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, EIP patterns and so on.

We then build our own custom Camel tooling to run in Apache Karaf and hawtio where we leverage information from the Camel catalog. We hope this chapter inspires you to start using some of these tools with your Camel development, and possibly also build custom tools yourself.

Lets get started with the traditional developer tools to develop Camel routes using a graphical editor.

19.1 Camel Editors

Building Camel tools for graphical Camel development is a very large task and could explain why we only see such tools from commercial vendors such as Red Hat and Talend. This book will cover the tool we know the best which is the tools from JBoss, Apache Camel and hawtio projects.

19.1.1 JBoss Tools for Apache Camel

The JBoss Tools for Apache Camel (formerly known as Fuse IDE) (<http://tools.jboss.org/features/apachecamel.html>) is a plugin that can be installed in Eclipse providing a set of Camel development capabilities such as:

- Graphical Camel editor
- Graphical Data Mapper
- Type-safe endpoint editor
- Integrated Camel tracer and debugger
- Easy deployment to application servers

The main course from the JBoss Tools for Apache Camel is the graphical editor. The editor works with Spring or Blueprint XML DSLs only. The tools are 100% decoupled from your source code and there is no vendor lock-in. You can open any Camel project in Eclipse and with this tool you can use its graphical editor to visualize the Camel routes and/or edit those routes.

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 the tool and at any point stop. These tools all operate on your project's source code *as-is* and don't use tooling specific meta data to be saved with your source code.

We will now look at some of the features this tool provides. You can find the sample project we use in the chapter19/fuse-ide directory.

GRAPHICAL CAMEL EDITOR

Figure 19.1 shows the graphical editor in action.

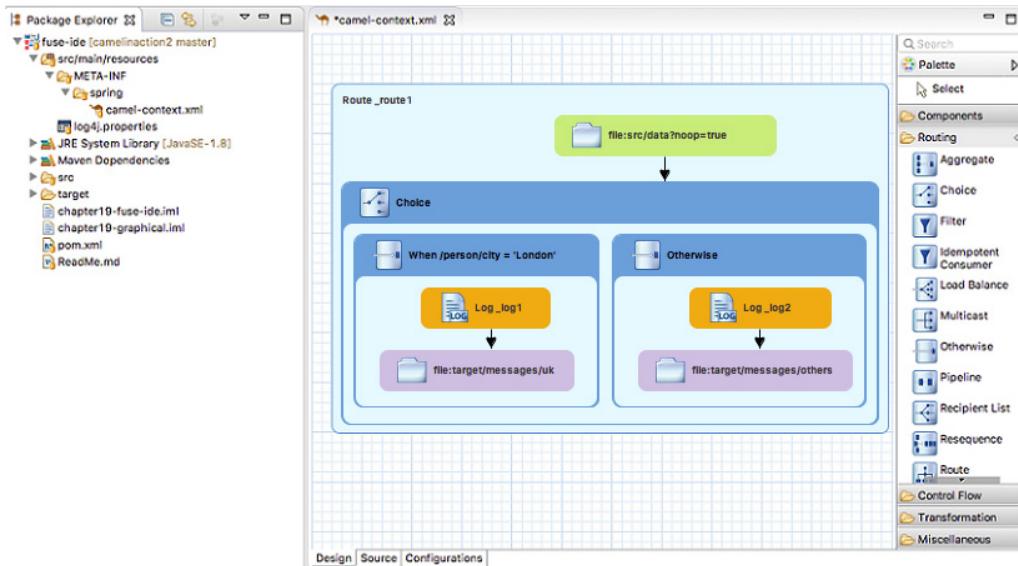


Figure 19.1 - Graphical Camel editor showing the Camel route using EIP icons and how they are connected. Clicking an EIP allows to configure the EIP in the properties panel in a type-safe way.

The Camel project which is loaded into Eclipse shown in figure 19.1 is a simple Camel route from a Spring XML file. You can click each of the EIPs in the route and edit its properties in a type-safe manner as shown in figure 19.2.

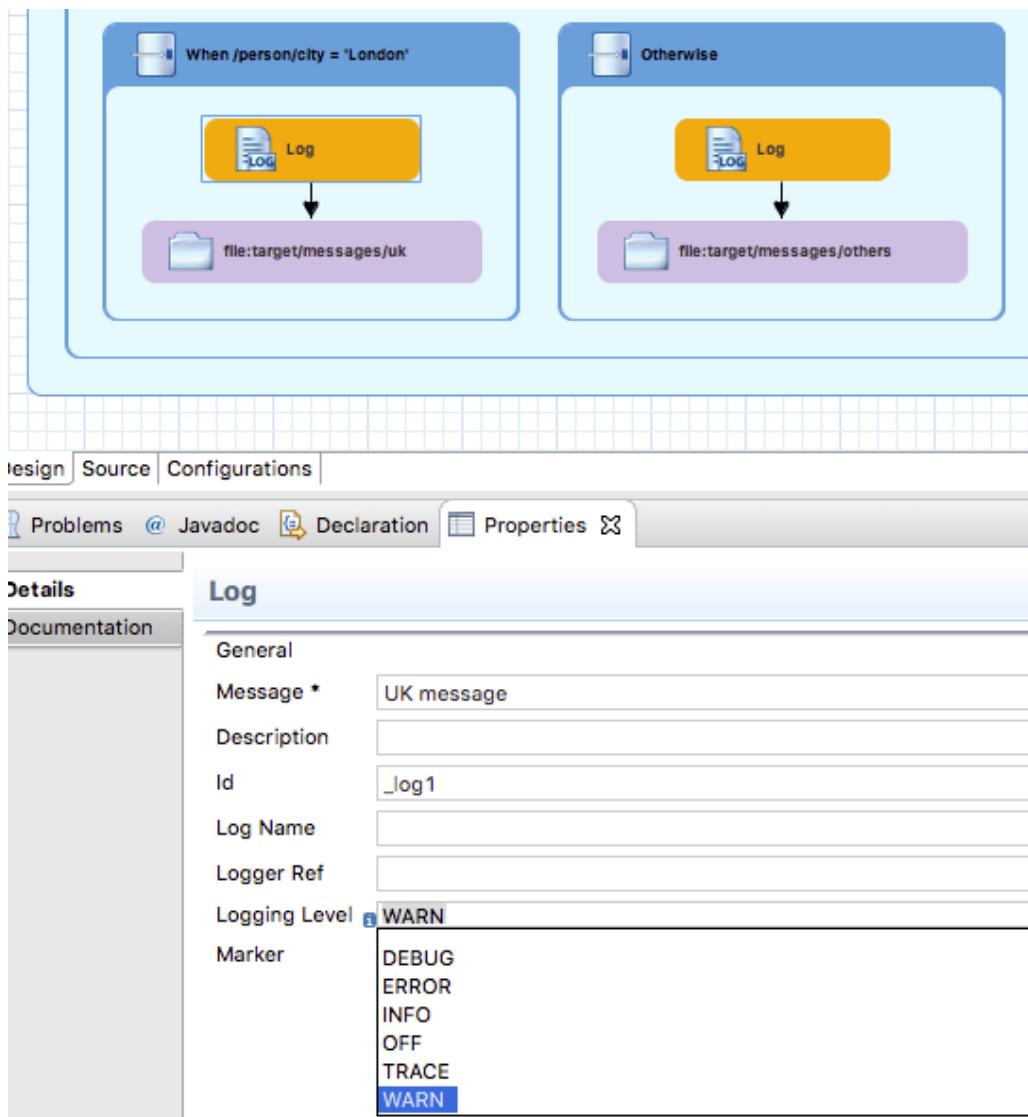


Figure 19.2 - Type safe editing the selected log EIP from the route in the panel below. The drop down panel for the logging level shows the possible values accepted.

In figure 19.2 we have selected the Log EIP which can be seen by the tiny yellow box around its borders. In the properties panel below the possible options from the Log EIP is shown with its current values. If we want to change the logging level we 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/graphical is a Camel route that consumes files from the following endpoint:

```
<from uri="file:src/data?noop=true"/>
```

Now suppose we want to configure a read lock on this endpoint. Using the tool you have all the options available presented in the editor and access to the documentation as a tooltip by hovering the mouse over the label of the option. Figure 9.3 shows how we are changing the endpoint to use the changed read lock using a 5 second interval.

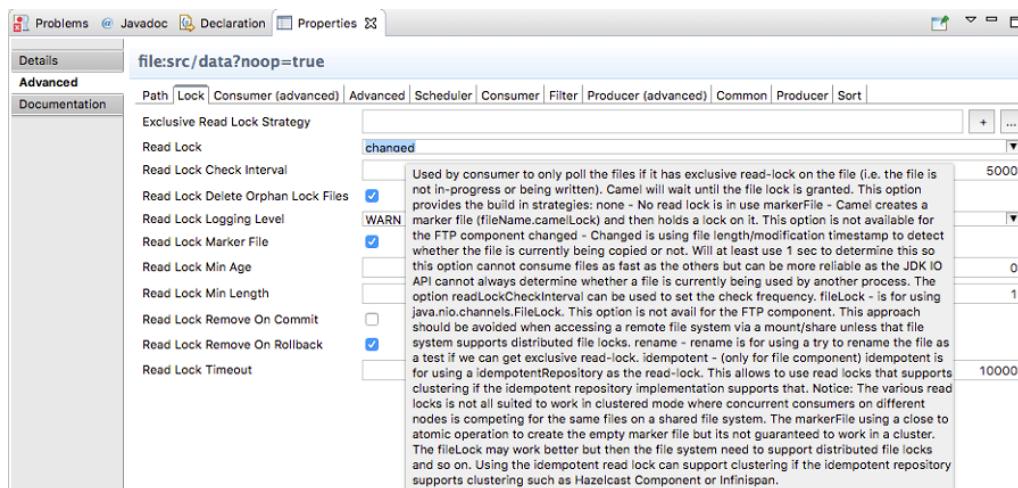


Figure 19.3 - Editing the Camel endpoint in a type-safe manner using the tooling. Configured the read lock option be changed, and documentation is shown in the tooltip when hovering the mouse over the row with the option.

After editing the endpoint is updated accordingly:

```
<from uri="file:src/data?
noop=true&readLock=changed&readLockCheckInterval=5000"/>
```

INTEGRATION CAMEL TRACER AND DEBUGGER

We previously covered various techniques for debugging Camel routes in chapter 8. The JBoss Tools for Apache Camel provides an Eclipse based debugger.

To try this in action, then 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 the breakpoint is set. Then right click the `camel-context.xml` file as shown in figure 19.4, and select *Debug As -> Local Camel Context (without tests)*.

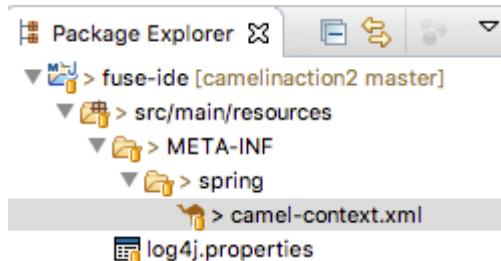


Figure 19.4 - To debug a Camel application, then right click the XML file containing the <camelContext> and chose Debug As -> Local Camel Context (without tests) to start the application in debug mode.

Eclipse will then switch to debug perspective, and when a message hits the breakpoint, the EIP icon will be highlighted using a dashed red border, as shown in figure 19.5.

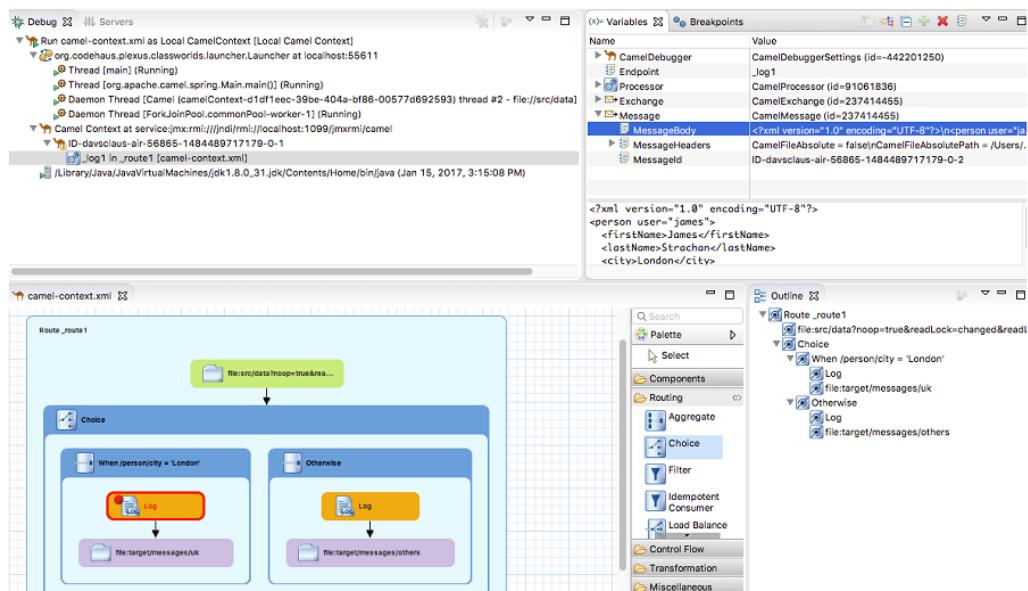


Figure 19.5 - Debugging Camel application in Eclipse where a Camel message hit the breakpoint highlighted by the red square. In the top righthand corner the Variables tab shows the content of the message. We have selected the message body which is shown below, which in this example is an XML message showing that James Strachan lives in London (he lived there at the time 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 are ready to continue, then click either

the resume or step over button. The latter allows 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 with the red dot in the top corner. When the breakpoint was hit we clicked the step over button (F6) and the debugger runs to the next EIP which is the file endpoint indicated by the red square.

The tools has more capabilities such as a basic data transformation mapper that allows you to visual map between common types such as XML, JSON, and Java. The mapper is using Dozer under the covers which means there is no vendor lock-in. You can always at any time stop using the visual data mapper and hand edit the dozer mapping files - and after all real developers like to be in control and not using graphical drag/drop tools that look good on demos, but fail when it comes to more complicated use-cases.

A summary of the JBoss tools for Apache Camel is listed in table 19.1.

Table 19.1 The good and not so good about the JBoss Tools for Apache Camel

| Pros | Cons |
|--|--|
| Graphical Representation of Camel routes | Only works with Eclipse |
| Powerful and great Camel debugger natively integrated in Eclipse | Can only edit and debug Camel routes created using XML DSL |
| 100% Open Source and Eclipse licensed | The graphical data mapper only provides basic mapping functionality |
| Type-Safe Camel endpoint editor | Non intuitive/easy installation procedure |
| Documentation for all components and EIPs included | Follows JBoss Fuse releases and may not always work with latest Apache Camel release |

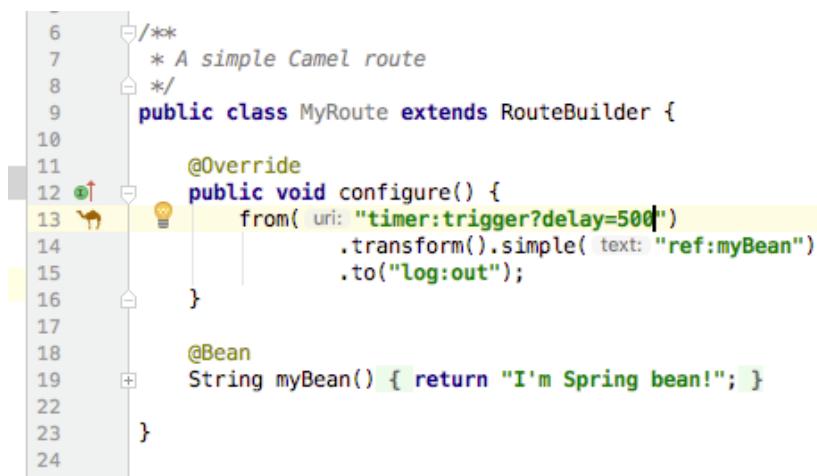
Lets look at another Camel editor that works in a different way.

19.1.2 Apache Camel IDEA Plugin

IntelliJ IDEA users should not miss out 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 better explained by just trying it out in action, so lets do that.

To install the plugin you started IDEA and open its preferences. From within Preferences you select Plugins and click Browse Repositories. In the search field you type Apache Camel to find the Apache Camel IDEA Plugin which you select and click Install. After installation you need to restart IDEA.

From the source code of the book you can open the chapter19/camel-idea-editor example in IDEA. Then open the Camel route file which is the MyRoute.java file and you should notice the Camel plugin appearance subtle 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
23
24

```

Figure 19.7 - In the gutter there is 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 showing a little Camel icon. For example position the cursor as shown in figure 9.7, 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 present the user with a list of possible endpoint options you can use to configure the endpoint as shown in figure 19.8.

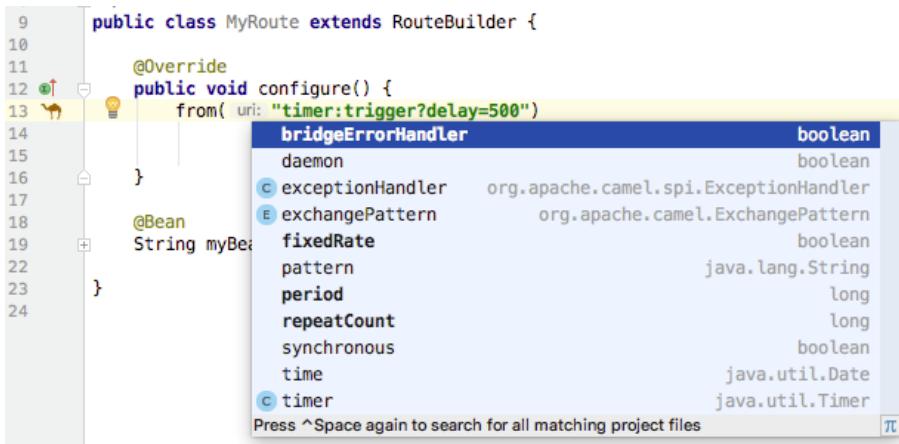


Figure 19.8 - Pressing **ctrl + space** shows a list of possible options that can be used to configure the endpoint. The options highlighted in bold are the most commonly used options.

The list works just as if you are 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 by the side which presents documentation for the currently selected option in the list. This gives you information at your fingertips, where you don't have to open a web browser and find the Camel component documentation. And in case you want to go there, then just press **ctrl + F1** and IDEA will open the appropriate web page for you in the browser.

You can also use **ctrl + space** when configuring the values of the endpoint options. Figure 19.9 shows an example where we are configuring the acknowledgement mode option on a JMS endpoint.

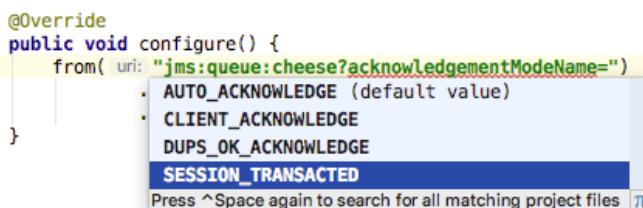


Figure 19.9 - Press **ctrl + space** while editing the value of an endpoint option can provide a list of possible choices if the option is an enum based type such as the JMS acknowledgement mode option.

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 red underline.

If this is too much for you, then the plugin can be configured from the preference, where you find Apache Camel in the Other Settings group.

The plugin works with all of the Camel DSLs so you can use this with Java, XML, Groovy, Scala and Kotlin. In other words just 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 as its capable of downloading the catalog (we will cover the Camel catalog in section 19.2) information from newer or older versions on the fly.

That completes our coverage of the Apache Camel IDEA Plugin and a summary is listed in table 19.2.

Table 19.2 The good and not so good about the Apache Camel IDEA tooling

| Pros | Cons |
|--|--|
| <ul style="list-style-type: none"> Integrates natively with IDEA in the code editor. Type safe and real time validation as you type in the code editor Works with any Apache Camel release Very easy to install 100% Open Source and Apache licensed Supports Java, XML, Scala, Groovy, and Kotlin | <ul style="list-style-type: none"> Works only on IDEA. But there are plans for Eclipse support in the future. No graphical visualization of Camel routes At this time of writing the plugin is still early in development |

Now we will step outside IDEA and other IDE editors for that matter and talk about a great little tool that comes out of the box with Apache Camel. It's the Apache Camel Maven plugin that is capable of scanning your source code and reporting invalid Camel endpoints.

19.1.3 Camel Validation using Maven

You may have experienced that starting your Camel application fails because of one or more Camel endpoint was misconfigured due to an invalid option. This is both a blessing and a curse with Camel, that endpoints are so quick and easy to configure using URI parameters. The endpoint URI is essentially a String type, such as the following file endpoint:

```
file:src/data?noop=true&recursive=true
```

Now suppose you mistyped 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}]
```

Would it not be great if we could validate those Camel URIs before you run your Camel application? Great news, the existing Camel Maven plugin from Apache Camel is capable of this.

The story of the fabric8 Camel Tooling

This validation functionality which we are about to cover was first developed as an experiment at the fabric8 (<https://fabric8.io/>) project. After more than a year in active development at fabric8 the plugin code was stable and then donated to Apache Camel for inclusion from Camel 2.19 onwards.

At the same time the fabric8 team experimented with a Camel editor which was based on JBoss Forge that allows 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. However we wanted tighter integration with the IDE editors than what JBoss Forge provides. On the even of 23rd December 2016 Claus created an experimental prototype of a Camel plugin for IDEA. This prototype was able to provide a list of possible Camel options you can configure on any Camel endpoint. In other words it provides type-safe smart completions directly in the IDEA editor, as you have for Java code. Over the next couple of months this prototype matured into the Camel IDEA plugin which will be covered in this book as well. Its planned for this plugin to be donated to Apache Camel as well when the code is mature and stable. However at this time of writing the code is hosted at github at: <https://github.com/davsclaus/camel-idea-plugin>

The accompanying source code for the book 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:2.19.0:validate (default) @ chapter19-camel-maven-validate ---
[INFO] Using Camel version: 2.19.0
[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)
```

As you can see there are two errors in the source code, the first error is the mistype of the recursive option. The tooling is giving you suggestions what what the option could be. The second error is in a log endpoint where we have configured the showAll option using a lower case a in all. The correct option is spelt showAll.

ADDING THE PLUGIN TO YOUR PROJECT

To use the validation plugin in your Camel projects add the following to the pom.xml

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <version>2.19.0</version>
</plugin>
```

And then you can run the validation goal using:

```
mvn camel:validate
```

You can also enable the plugin to automatic run as part of the build, but configuring 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>2.19.0</version>
  <executions>
    <execution>
      <phase>process-classes</phase>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

By default the plugin only validates the source code 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 in bold:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <version>2.19.0</version>
  <executions>
    <configuration>
      <includeTest>true</includeTest>
    </configuration>
    <execution>
      <phase>process-classes</phase>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

The plugin has a number of options you can configure, which you can find details in the Camel plugin documentation at the Camel website.

All of the tools we have covered so far: JBoss tools for Apache Camel, IDEA Camel Plugin, and the Camel Maven Plugin are all using 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 what's included in the release. The catalog is shipped in a independent standalone camel-catalog JAR containing the following information:

- List of all components, data formats, languages, and EIPs in the release
- JSON schema with extensive details for every option all above supports
- Human readable documentation for every option all above supports
- Categorization of all above (eg find all NoSQL components)
- XML Schema for the XML DSL
- Maven Archetype Catalog of all the Camel archetypes
- Entire website documentation as ASCII doc and HTML files
- Validator for Camel endpoint and Simple language
- Java, JMX, and REST API

As you can see the catalog provides a wealth of different kind of information that tooling can tap into and leverage. For example the JBoss tools for Apache Camel uses this in the graphical editor to know all details about every EIP pattern 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 endpoint found while scanning the source code.

The camel-catalog JAR includes the information using the following directory layout:

```
org
└── apache
    └── camel
        └── catalog
            ├── archetypes      (Maven archetype catalog)
            ├── components      (JSON schema)
            ├── dataformats     (JSON schema)
            ├── docs            (Ascii and HTML docs)
            ├── languages        (JSON schema)
            ├── models           (JSON schema)
            └── schemas          (XML schema)
```

Each of these directories 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 listing 19.1.

Listing 19.1 - JSON schema of the timer component from the camel-catalog

```
{
  "component": {
    "kind": "component",
    "scheme": "timer",
    "syntax": "timer:timerName",
    "title": "Timer",
    "description": "The timer component is used for generating message exchanges when a timer fires.",
    "label": "core,scheduling",
    "deprecated": "false",
    "async": "false",
    "consumerOnly": "true",
    "javaType": "org.apache.camel.component.timer.TimerComponent",
    "groupId": "org.apache.camel",
    "artifactId": "camel-core",
    "version": "2.19.0"
  },
  "componentProperties": {
  },
  "properties": {
    "timerName": { "kind": "path", "group": "consumer", "required": true },
    "bridgeErrorHandler": { "kind": "parameter", "group": "consume" },
    "delay": { "kind": "parameter", "group": "consumer", "type": "fixedRate" },
    "fixedRate": { "kind": "parameter", "group": "consumer", "type": "period" },
    "period": { "kind": "parameter", "group": "consumer", "type": "repeatCount" },
    "repeatCount": { "kind": "parameter", "group": "consumer", "type": "exceptionHandler" },
    "exceptionHandler": { "kind": "parameter", "group": "consumer" },
    "daemon": { "kind": "parameter", "group": "advanced", "label": "Advanced consumer options" },
    "exchangePattern": { "kind": "parameter", "group": "advanced", "label": "Advanced exchange pattern" },
    "pattern": { "kind": "parameter", "group": "advanced", "label": "Advanced pattern" },
    "synchronous": { "kind": "parameter", "group": "advanced", "label": "Advanced synchronous" },
    "time": { "kind": "parameter", "group": "advanced", "label": "Advanced time" },
    "timer": { "kind": "parameter", "group": "advanced", "label": "Advanced timer" }
  }
}
```

- ➊ The name of the component
- ➋ URI syntax (without parameters)
- ➌ Labels are used for categorizing the component
- ➍ This component can only be used as a consumer (eg from)
- ➎ The Maven coordinate for the JAR that contains the component
- ➏ Component level options
- ➐ Endpoint level options (text abbreviated)

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 ➊ and syntax ➋ . Then labels are used to categorize the component ➌ . This component has the labels core and scheduling.

In Camel a component can be used as a consumer, producer or both. The JSON schema

includes this information if the component can only be a consumer or producer by consumerOnly ④ or producerOnly. If neither of those information exists then the component can be used as both. There is also information which JAR file contains the component as Maven coordinates ⑤ . The last two parts are documenting each option you can configure on the component level ⑥ and endpoint level ⑦ . 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 which has been abbreviated in the listing. The following is a snippet of the delay option in its entirety:

```
"period": { "kind": "parameter", "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)." },
```

Lets break down the option and explain each detail as listed in table 9.3.

Table 9.3 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 an URI query parameter. |
| 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 an 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 patterns, data formats and languages 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 may include more details in future Camel releases. And therefore the JSON Schema may change and include additional information. However the basic structure as shown here in listing 19.1 is expected to stay.

If you want to write some custom Camel tooling then the camel-catalog is a great source of information. For example suppose you want to write a Camel tool that can report whether running Camel applications is using deprecated Camel components.

19.2.1 Developing custom tooling using the camel-catalog

At Rider Auto Parts you have been using Camel for a long time. You did your first steps with Camel in 2011 when the 1st edition of this book was published. Over the years you have created and put many different Camel applications into production. Now you would like to build some custom tooling to report utilization and statistics of these applications. You decide to start easy and build a tool that reports usage of deprecated Camel components. You plan to add more checks for *old stuff* as you call it.

In this section we will show you how to write such an *old stuff* tool. The tool will be built to run as a command in Apache Karaf, and in section 19.3.3 we will build the tool to run in a web browser as a hawtio plugin.

To find all the running Camel applications in the JVM we use JMX to find their `ObjectNames` as shown below:

```
public static Set<ObjectName> findCamelContexts() throws Exception {
    MBeanServer server = ManagementFactory.getPlatformMBeanServer();
    return server.queryNames(
        new ObjectName("org.apache.camel:type=context,*"), null);
}
```

The returned `ObjectName` refers to the JMX API of `CamelContext` which is defined in the `org.apache.camel.api.management.mbean.ManagedCamelContextMBean` interface. This interface has the method `findComponentNames` which returns the names of the Camel components that are in use. So finding the components is a matter of using the following few lines of Java code:

```
public List<String> findComponentNames(ObjectName on) throws Exception {
    MBeanServer server = ManagementFactory.getPlatformMBeanServer();
    return (List<String>) server.invoke(on,
        "findComponentNames", null, null);
}
```

With the list of component names the Camel application uses in our hand, we can then ask the Camel catalog which of these components are deprecated. How to do this is shown in listing 19.2.

Listing 19.2 - Method to check if a Camel component is deprecated or not

```
public List<String> findDeprecatedComponents() throws Exception {
    List<String> answer = new ArrayList<>();

    Set<ObjectName> camels = findCamelContexts(); ①
    for (ObjectName on : camels) {
        List<String> names = findComponentNames(on); ②
        for (String name : names) {
```

```

        if (isDeprecatedComponent(name)) { ③
            answer.add(name);
        }
    }
    return answer;
}

private boolean isDeprecatedComponent(String name) {
    String json = catalog.componentJJsonSchema(name); ④

    List<Map<String, String>> rows =
        JsonSchemaHelper.parseJsonSchema("component", json, false); ⑤
    for (Map<String, String> row : rows) {
        if (row.get("deprecated") != null) {
            return "true".equals(row.get("deprecated")); ⑥
        }
    }
    return false;
}

```

- ① Find all Camel applications running in the JVM
- ② Find all the components the Camel application is using
- ③ Check if the component is deprecated
- ④ Load the JJson Schema of the component from the Camel catalog
- ⑤ Parse the JJson into a row structure
- ⑥ Find the deprecated row and check if its value is true

In listing 19.2, the first method is the public method that performs the task of finding all the names of the Camel components in use that has been deprecated. The method first find all the running Camel applications ① and then for each Camel application find the list of components it is using ② , and for each component it calls the second method ③ to check if its deprecated or not. The second method uses the Camel catalog to load the JJson schema of the component ④ because this schema contains the information whether the component is deprecated or not. To find that information we need to parse the component JJson schema. We could use a JJson library like Jackson but instead we are using the built-in `JsonSchemaHelper` class from the `camel-catalog` ⑤ , which parses that data into a row structure as a List of Map. We then loop the List and find the deprecated row from the Map and check whether its value is true ⑥ .

The source code of the book contains this custom tooling in the `chapter19/custom-tooling` directory.

To use this tool we would like to build a custom Karaf command which we can install in our Apache Karaf/ServiceMix application servers.

19.2.2 Camel tooling with Karaf

In this section we will build a custom Karaf command that runs the deprecation checker we build in the last section. Apache Karaf allows to install custom commands which you build and

install as an OSGi bundle. To build a custom Karaf command you need to add the following dependency to your Maven pom.xml file:

```
<dependency>
    <groupId>org.apache.karaf.shell</groupId>
    <artifactId>org.apache.karaf.shell.console</artifactId>
    <version>4.0.8</version>
</dependency>
```

This dependency provides the needed classes to build custom Karaf commands. Listing 19.3 shows the code for the deprecated checker.

Listing 19.3 Karaf command to run the Camel deprecated component checker

```
package camelinaaction.karaf;

import java.util.List;
import camelinaaction.DeprecatedValidator;
import org.apache.felix.gogo.commands.Command;
import org.apache.karaf.shell.console.OsgiCommandSupport;

@Command(scope = "oldstuff", name = "deprecated-components",
          description = "Lists all deprecated components in use.")      ①
public class DeprecatedComponentsCommand extends OsgiCommandSupport { ②

    protected Object doExecute() throws Exception {
        DeprecatedValidator validator = new DeprecatedValidator();
        List<String> names = validator.findDeprecatedComponents();      ③
        if (names.isEmpty()) {
            System.out.println("No deprecated Camel components found."); ④
        } else {
            System.out.println("Deprecated Camel components found:");
            for (String name : names) {
                System.out.println("\t" + name);
            }
        }
        return null;                                                 ⑤
    }
}
```

- ① @Command annotation configuring command name and description
- ② Extend base class for command support
- ③ Use the validator to check for deprecated Camel components
- ④ Output whether any deprecated components was found or not
- ⑤ Return null

A Karaf command is a class annotated with `@Command` ① where you specify the scope (scope is used to group commands together) and name, and description of the command. The class must then extend the `OsgiCommandSupport` class ② and implement the logic in the `doExecute` method. Because we have already built the logic to find all the Camel component names that has been deprecated we can implement the logic in two lines of code ③ . To make the command output in the shell you simply just use `System.out.println` to print to the console ④ . The command should then return `null` to complete.

You then need to register this command in a blueprint XML file located in OSGI-INF/blueprint directory so Karaf can detect the command. This is done using the following lines of XML:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
    <command-bundle xmlns="http://karaf.apache.org/xmlns/shell/v1.0.0">
        <command name="oldstuff/deprecated-components">
            <action class="camelinaction.karaf.DeprecatedComponentsCommand"/> ①
        </command>
    </command-bundle>
</blueprint>
```

① Class name of our command

You can find this command in the source code in chapter19/custom-tooling-karaf directory. The source code has a readme.md file with instructions how to install the custom command in Karaf/ServiceMix. The source code also contains a sample Camel application which runs the following Camel route:

```
<route>
    <from uri="quartz:foo/bar?cron=0/5+*+*+*+?" />
    <setBody>
        <method ref="helloBean" method="hello"/>
    </setBody>
    <log message="The message contains ${body}"/>
    <to uri="mock:result"/>
</route>
```

If you install the sample application and run the custom command from Karaf to check for deprecated Camel components in use as shown:

```
karaf@root()> oldstuff:deprecated-components
Deprecated Camel components found:
    quartz
```

Here we can see the command has detected that the quartz component is deprecated. If you change the sample application to use quartz2 component then the command reports no usage of deprecated components.

Lets leave Karaf and move on to the web and look at Camel tooling with hawtio.

19.3 hawtio - A web console for Camel and Java applications

So what is hawtio (<http://hawt.io>) and what does it do? Here is what hawtio say what it does:

It's a pluggable management console for Java stuff which 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 have read this book in chronological order then you may remember we have talked about hawtio in previous chapters, such as chapter 16 where we briefly covered using hawtio to manage Camel application.

In this section we will take a deeper look at hawtio and unlock some of the features it provides that is relevant to Camel, and at the end build a custom hawtio plugin that runs the deprecated component checker we have previous build.

19.3.1 Overview of hawtio functionality

Hawtio provides the following Camel features out of the box:

- Lists all running Camel applications in the JVM
- Detailed information of each Camel application such as version number, runtime statics
- Lists of all routes in each Camel applications and their runtime statistics
- Manage the lifecycle of all Camel applications and their routes, so you can restart / stop / pause / resume, etc.
- Graphical representation of the running routes along with real time metrics
- Embedded documentation of in use components, endpoints, and EIP patterns
- Live tracing and debugging of running routes
- Profile the running routes with real time runtime statics; detailed specified per processor
- Update routes using a text based XML editor (not persistent update)
- Browsing and sending messages to Camel endpoint(s)

As you can see hawtio provides many Camel features out of the box. The most prominent Camel feature of hawtio is likely the Camel route diagram. The diagram in figure 19.10 is the sample Camel application from chapter19/deprecated-component-karaf which we have installed in Karaf.

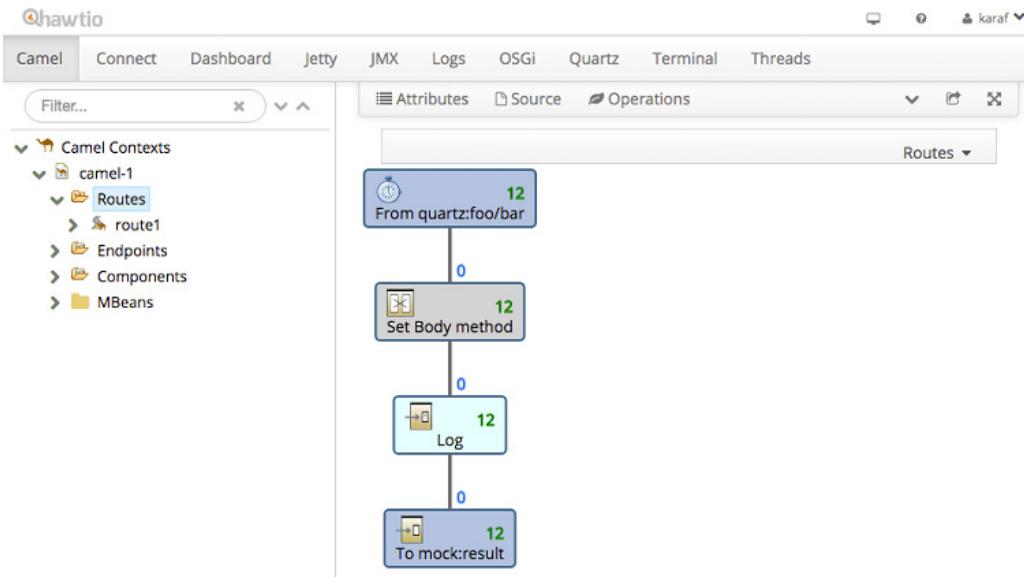


Figure 19.10 - Visual representation of the running Camel routes using the hawtio web console. The green numbers are the messages processed, and the blue number is the number of messages that are inflight (currently being processed). The numbers updates in real time.

Installing hawtio in Karaf

You can install hawtio in Apache Karaf/ServiceMix using the following commands:

```
karaf@root()> feature:repo-add hawtio 1.4.68
Adding feature url mvn:io.hawt/hawtio-karaf/1.4.68/xml/features
karaf@root()> feature:install hawtio
```

Then you can access hawtio in the web browser at: <http://localhost:8181/hawtio> and login with karaf/karaf (default username and password)

hawtio includes a route debugger, so lets try this in action.

19.3.2 Debugging Camel routes using hawtio

When developing Camel routes it can be handy to be able to debug at the route level. hawtio comes with a route debugger, which you can run from a web browser. Lets 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 5th second. Then a random number between 0 and 1000 is generated as the message body, and depending whether the number is a high or low number routed accordingly using the Content Based Router EIP. This example is provided in the chapter19/hawtio-debug directory which you can run from Maven using:

```
mvn compile exec:java
```

Because we want to debug the route, we can start the example with hawtio embedded. This can be done by running the following Maven goal:

```
mvn compile hawtio:run
```

Then what happens is hawtio is started from the Maven plugin which then starts the Camel application using a Java main class. The Java main class is being configured in the Maven pom.xml file as shown in bold:

```
<plugin>
<groupId>io.hawt</groupId>
<artifactId>hawtio-maven-plugin</artifactId>
<version>1.4.68</version>
<configuration>
<mainClass>camelinaction.MainApp</mainClass>
</configuration>
</plugin>
```

When you run the `hawtio:run` goal then hawtio and the Camel application are started together in the same JVM. And after a little while hawtio opens automatic 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 number of messages being processed.

Okay lets get on with it, how do you debug a Camel route using hawtio? To do this you need to select the route you want to debug in the tree on the left hand side, and because there is only one route in this example you should select `route1`. Then in the center of the screen there is a `Debug` button you should click. 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. Then after a little while the EIP should display a blue ball which indicate a breakpoint was hit. In the bottom of the screen the message is listed which you can click to show its message body and headers. After doing this you should have a similar screen as shown in figure 19.11.

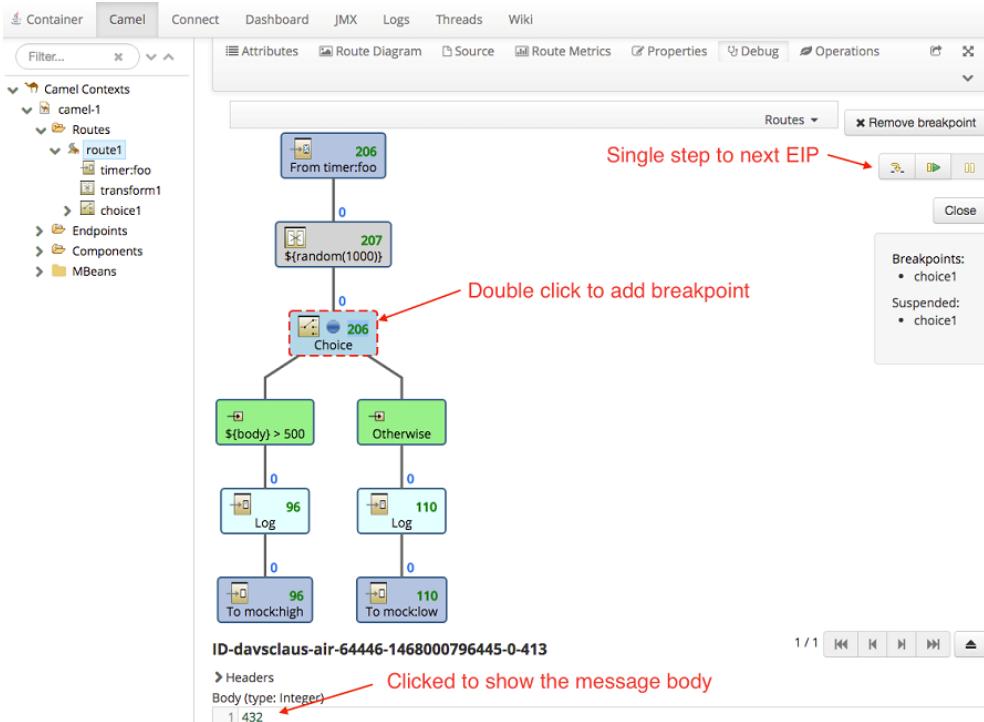


Figure 19.11 - Using hawtio to debug Camel route. Double click an EIP to set a breakpoint. When a breakpoint is hit the ball changes to blue color, and in the bottom of the screen you can see the current message body and its headers. On the top right hand you can click the single step button to continue routing and pause the debugger at the next EIP.

In figure 19.11 the debugger has paused at the Content Based Router EIP (choice) which is indicated by the blue ball. In the bottom of the screen we can see the message body (you need to click the message to expand to show the content of the message) has the value of 432. So when we click the single step button in the top right corner, the message is routed and paused at the next EIP which will be the otherwise branch in the Content Based Router; because 432 is not higher than 500. When you are done with the debugger remember to click the Close button.

Debugging API

The debugger in hawtio and the Eclipse based JBoss Tools for Apache Camel are both using 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, single step, update the message body/headers, etc.

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 startup hawtio in standalone mode and then from hawtio connect to the existing JVM. In the example instead of running the `hawtio:run` plugin, you run `exec:java` instead

```
mvn exec:java
```

And then from another terminal you download hawtio-app and run it as a standalone Java application:

```
java -jar hawtio-app-1.4.68.jar
```

When hawtio opens in the web browser, then the Connect plugin should be default selected. 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. We have done this and you should have a screen similar to figure 19.12.

| PID | Name | Agent URL |
|-------|--|---|
| 38626 | | |
| 38660 | IntelliJ IDEA | |
| 39193 | hawtio | |
| 39135 | org.codehaus.plexus.classworlds.launcher.Launcher exec:java | http://127.0.0.1:65318/jolokia/ |

Figure 19.12 - Running hawtio in standalone mode, to connect to any local running JVMs on your computer. When clicking the hyperlink in the Agent URL hawtio opens a web page connected to that JVM which allows you to manage and look inside that running JVM such as the Camel plugin if Camel is running.

TIP hawtio is a great web console that can visualize and present many informations about your running Camel applications. We recommend you take a look at this tool.

Okay we have one more thing to cover with hawtio, how to build a custom plugin so you can run the tooling we previously developed to check for usage of deprecated Camel components.

19.3.3 Building hawtio plugins

In this section you will learn how to build a custom plugin for hawtio.

A custom hawtio plugin is built as Maven project and packaged as a Java Web application.

In the Maven pom.xml file you need as minimum the following two dependencies:

```
<dependency>
    <groupId>io.hawt</groupId>
    <artifactId>hawtio-plugin-mbean</artifactId>
    <version>1.4.68</version>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>
```

The former dependency is used for bootstrapping the custom plugin in hawtio using JMX. The latter dependency is the Servlet API because the plugin is deployed as a Java Web application. The plugin source code is a set of Java and JavaScript/TypeScript code which is organized as a Java web application structure using the following directory layout:

```
src
└── main
    └── java           (Java source code)
        └── camelinaction
    └── resources
        └── WEB-INF      (web.xml file)
    └── webapp
        └── plugin       (html/css and javascript files)
            ├── css
            ├── doc
            ├── html
            └── js
```

The source code is divided into backend and frontend:

- Java code for the backend
- HTML and javascript for the frontend

In the backend you expose the APIs and services the frontend needs. So lets start there.

BUILDING JAVA CODE FOR THE BACKEND

A hawtio plugin needs Java code to bootstrap the plugin. This is done using a Java Servlet ContextListener as shown in listing 19.4.

Listing 19.4 ContextListener to bootstrap custom hawtio plugin

```

import io.hawt.web.plugin.HawtioPlugin;

public class PluginContextListener implements ServletContextListener { ①
    private HawtioPlugin plugin;
    private CamelComponentDeprecated deprecated;

    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent) {
        ServletContext context = servletContextEvent.getServletContext();

        plugin = new HawtioPlugin();
        plugin.setContext(context.getInitParameter("plugin-context")); ②
        plugin.setName(context.getInitParameter("plugin-name"));
        plugin.setScripts(context.getInitParameter("plugin-scripts"));
        plugin.setDomain(null);
        plugin.init();

        deprecated = new CamelComponentDeprecated(); ③
        deprecated.init();
    }

    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        deprecated.destroy(); ④
        plugin.destroy(); ④
    }
}

```

- ① ServletContextListener used to bootstrap the plugin in hawtio
- ② Register custom plugin using HawtioPlugin
- ③ Startup custom plugin in hawtio and the service needed for checking for deprecated components
- ④ Stop the services when context is being stopped

Listing 19.4 shows the typical Java code you need to write to bootstrap the custom plugin in hawtio. The code worth to mention is `HawtioPlugin` ② is a class from hawtio which is used to register custom plugins. In addition you need to build backend service(s) and register during the bootstrap as well ③ . This plugin will use `CamelComponentDeprecated` ③ as the service, as shown in listing 19.5.

The backend services are exposes as JMX MBeans. This is done by creating an interface which name ends with `MBean` as shown:

```

public interface CamelComponentDeprecatedMBean {
    List<String> findDeprecatedComponents();
}

```

The implementation of the service implements this interface, and methods to register and unregister itself in the JMX MBean registry. Listing 19.5 shows how this can be done:

Listing 19.5 - Implementation of backend service as JMX MBean

```

public class CamelComponentDeprecated implements
    CamelComponentDeprecatedMBean {
    private MBeanServer mBeanServer;
    private ObjectName objectName;
    private DeprecatedValidator validator = new DeprecatedValidator();

    public void init() throws Exception { ①
        objectName = getObjectName();
        mBeanServer = ManagementFactory.getPlatformMBeanServer();
        mBeanServer.registerMBean(this, objectName);
    }

    public void destroy() throws Exception { ②
        mBeanServer.unregisterMBean(objectName);
    }

    protected ObjectName getObjectName() throws Exception { ③
        return new ObjectName("hawtio:type=CamelComponentDeprecated");
    }

    public List<String> findDeprecatedComponents() { ④
        try {
            return validator.findDeprecatedComponents();
        } catch (Exception e) {
            // ignore
        }
        return null;
    }
}

```

- ① Register this service as a JMX MBean
- ② Unregister this service from JMX
- ③ The JMX MBean ObjectName to use
- ④ Implementation of service to find deprecated Camel components

The bulk of the source code in listing 19.5 is code to register and unregister the service as a JMX MBean in the JMX platform server ① ② . The name the service used in the JMX tree is defined in the `getObjectName` method ③ . This name is important to remember as the frontend must use this name when calling the service.

The last task to do in the backend is to setup the web.xml file to use the ContextListener from listing 19.4 to bootstrap the plugin as shown:

```

<listener>
    <listener-class>camelinaction.PluginContextListener</listener-class>
</listener>

```

In `web.xml` you also need to configure a few `context-param` parameters specifying the name and context path of the plugin. Because you can externalize these values in the root `pom.xml` of the plugin you can use placeholders to refer to these values as shown:

```

<context-param>
    <param-name>plugin-context</param-name>

```

```

<param-value>${plugin-context}</param-value>
</context-param>
<context-param>
    <param-name>plugin-name</param-name>
    <param-value>${project.artifactId}</param-value>
</context-param>

```

In the Maven `pom.xml` file we specify these values under the `<properties>` section as shown:

```

<plugin-context>/oldstuff-plugin</plugin-context>
<plugin-name>oldstuff</plugin-name>

```

Yeah naming things in IT is hard, so you named the plugin *oldstuff*.

Lets move on to another world, some may say its a more scary world, the world of frontends where JavaScript rule the planet.

BUILDING JAVASCRIPT AND HTML FOR THE FRONTEND

The frontend is implemented using HTML and JavaScript/TypeScript so brace yourself for some fun times ahead. To build a quick prototype we have chosen to keep the HTML code simple and use a good old fashioned button to call the service as shown in listing 19.6.

Listing 19.6 - HTML code as the user interface for the custom plugin

```

<div class="oldstuff-controller" ng-controller="Oldstuff.OldstuffController"> ①
    <div class="dialog-body">
        <div class="buttonBar">
            <button class="btn btn-primary"
                   ng-click="findDeprecatedComponents()">Run check</button> ②
        </div>
    </div>

    <div class="row-fluid" ng-show="output">
        <div ng-repeat="component in output">
            {{component}}
        </div>
    </div>
</div>

```

- ① Name of controller with the JavaScript logic used in this HTML page
- ② Button that calls the backend service to find deprecated Camel components
- ③ Output the list of found deprecated components

The frontend technologies in hawtio are using the popular web framework AngularJS and Bootstrap for CSS styling.

In every HTML page you need to access business logic you need to use a controller containing this logic ①. To scan for deprecated components you click a button that calls the `findDeprecatedComponents` function ② which is defined in the controller. The found components is then listed row by row ③ using `ng-repeat` from AngularJS.

The controller ① is implemented in JavaScript (you can also use TypeScript) in the js/oldstuffPlugin.js file. Listing 19.7 do not show the entire source code but the most relevant part to know for plugin developers.

Listing 19.7 - JavaScript code as the controller for the custom plugin

```
var Oldstuff = (function(Oldstuff) {

    Oldstuff.pluginName = 'oldstuff_plugin';
    Oldstuff.mbean = "hawtio:type=CamelComponentDeprecated"; ①

    workspace.topLevelTabs.push({
        id: "oldstuff",
        content: "Oldstuff",
        title: "Plugin to check for old stuff",
        isValid: function(workspace) { return true; },
        href: function() { return "#/oldstuff_plugin"; },
        isActive: function(workspace) {
            return workspace.isLinkActive("oldstuff_plugin");
        }
    });
}

$scope.findDeprecatedComponents = function() { ③
    jolokia.request({
        type: 'exec',
        mbean: Oldstuff.mbean,
        operation: 'findDeprecatedComponents'
    }, onSuccess(render, {error: renderError}));
};

function render(response) { ④
    $scope.output = response.value;
    if ($scope.output.length === 0) {
        $scope.output = "No deprecated components found";
    }
    Core.$apply($scope);
} ⑤

hawtioPluginLoader.addModule(Oldstuff.pluginName); ⑥
```

- ① The name of the JMX MBean from the backend service
- ② To add a button to access the plugin in the top navigation bar
- ③ Function to find deprecated Camel components being used
- ④ Render the result from calling the function
- ⑤ Force update the web page in the web browser so the result is shown immediately
- ⑥ Install the custom plugin in hawtio

To access the backend service the controller must call the JMX MBean using the name it was registered in the JMX server ① . You should notice this name must be identical to the name we configured in listing 19.5. We then add the plugin to the hawtio navigation bar ② so we can click on the *Oldstuff* button to access the plugin. When you click the button in the HTML page it calls the function `findDeprecatedComponents` ③ that uses Jolokia to call the backend service. The result of this call is then rendered ④ and forcing the web page to be updated ⑤

so the result is shown immediately. And the last line registers the plugin with the hawtio plugin loader ⑥ .

We have now built the plugin and are ready to install and run the plugin in hawtio.

INSTALLING CUSTOM PLUGIN IN HAWTIO

We have provided this plugin with the source code for the book in the chapter19/custom-hawtio-plugin directory. You can find a readme.md file with instructions how to build and run the plugin. In this book we will show you how to build and run the plugin in Apache Karaf which we used in section 19.2.2.

First you need to build the plugin using Maven

```
mvn clean install
```

... then in the Karaf shell you install the plugin using:

```
install -s mvn:com.camelinaction/chapter19-hawtio-custom-plugin/2.0.0/war
```

Then from a web browser open the hawtio web console using the following url:

```
http://localhost:8181/hawtio
```

... and login with karaf/karaf (default username and password)

In the top navigation bar the plugin should be present with the name *Oldstuff*. When clicking the button in this plugin it should report whether any deprecated Camel components are in use, as illustrated in figure 19.13.

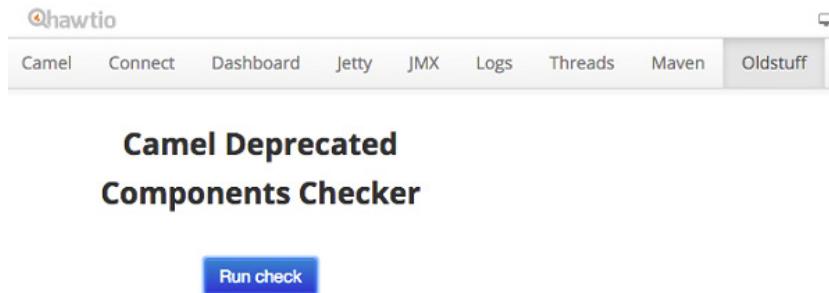


Figure 19.13 - Installed the custom hawtio plugin named Oldstuff. By clicking the Run check button the plugin reported the deprecated quartz component is in use

As you can see the custom plugin is installed seamless in hawtio and appears as a first class plugin in the navigation bar. The visual look and feel of the plugin is very simple and could need some CSS styling to look like a modern web application. Also the plugin could automatic run the check when the plugin is selected instead of having to click a button. And you could

improve the tool to report the Camel id which uses the deprecated components. We leave all these ideas up as exercise to the reader.

hawtio v1 vs v2

In this example we are using hawtio v1. A v2 is in development, so you may say why are we not using this version? A reason is that v2 currently is aimed as a web console toolkit for the fabric8 project based on Kubernetes and Docker technologies. The project has not been focusing on being a Java web console and are yet to bring over these parts from v1 to the v2 release. Migrating from v1 to v2 should mostly affect the frontend, as the backend is using Jolokia and JMX which should stay the same.

If you are interested in building web tooling for Java applications we encourage you to look into hawtio and the technologies we are 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 last chapter of *Camel in Action*, we've shown you the best Camel tools you can find on the internet. The most known tools are the Camel editors for graphical drag'n'drop development, such as the Eclipse tooling from JBoss.

However there is a set of smaller and lighter tools on the rise, such as the Apache Camel IDEA plugin which focus on being extending the source code editor instead of graphical drag'n'drop development. It is expected that some of these capabilities will find its way into Eclipse as well.

You learned that all the Camel components, data formats, EIPs and languages 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 should also not miss out the Camel Maven plugin that is able to check the source code during compile time to catch mis-configured Camel endpoints.

In the last third of this chapter we showed you how to build custom Camel tools such as a tool to check if any running Camel application uses deprecated components. You learned how to build this tool to run in Apache Karaf and in the hawtio web console. But these are not your only choices, you could also make tools as Maven Plugin, IDEA or Eclipse Plugins.

The takeaway from this chapter:

- *Graphical Tooling* - Many experienced camel developers prefer to work directly on the code, but we've covered these tools for new developers or those who prefer to work using graphical dag'n'drap editing.
- *Camel route debugger* - A very powerful feature from the JBoss Eclipse tooling is the Camel route debugger which is integrated seamless into Eclipse. An alternative debugger would be to use hawtio from a web browser.

- *Type-safe endpoint editor* - The Camel IDEA tooling provides a light-weight that allows you to edit/add endpoints directly in the source code from the cursor position. We think this kind of tooling would appeal to both new and experienced Camel developers.
- *Camel Maven Plugin* - Don't miss out the Maven plugin you can run during compilation to check for mis configured 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 does that.

With the goldmine of information from the Camel Catalog and the powers from Jolokia, Maven, IDEA, Eclipse, and hawtio we think you have powerful tools to build custom tools you may need for your Camel based integration applications.

These tools are all open source and have a very open community, so we welcome you to participate and help improved these tools with features you may find useful, or if you want to have some fun with hacking on code that is not your typical day to day job. Hope to see you around in the Camel communities.

Congratulations you have now read that last chapter of this book. We hope you enjoyed this second *Camel in Action* ride with us.

Regards

Claus Ibsen and Jonathan Anstey