

Compliments of  RED HAT
DEVELOPER
PROGRAM

Selections from



Camel in Action
Second Edition

Claus Ibsen and Jonathan Anstey

 manning



*Selections from
Camel in Action, Second Edition*

by Claus Ibsen and Jonathan Anstey

Chapter 1

Chapter 2

Chapter 3

Copyright 2018 Manning Publications
To pre-order or learn more about these books go to www.manning.com

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

For more information, please contact


Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2018 by Manning Publications Co. All rights reserved.

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

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

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

 Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haines

ISBN 9781617295737
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 23 22 21 20 19 18

brief contents

Part 1 First steps 1

1 MEETING CAMEL 3

- 1.1 Introducing Camel 4
- 1.2 Getting started 10
- 1.3 Camel's message model 15
- 1.4 Camel's architecture 18
- 1.5 Your first Camel ride, revisited 24
- 1.6 Summary 25

2 ROUTING WITH CAMEL.....27

- 2.1 Introducing Rider Auto Parts 28
- 2.2 Understanding endpoints 29
- 2.3 Creating routes in Java 33
- 2.4 Defining routes in XML 39
- 2.5 Endpoints revisited 49
- 2.6 Routing and EIPs 55
- 2.7 Summary and best practices 71

Part 2 Core Camel 73

3 TRANSFORMING DATA WITH CAMEL.....75

- 3.1 Data transformation overview 76
- 3.2 Transforming data by using EIPs and Java 77
- 3.3 Transforming XML 87
- 3.4 Transforming with data formats 91
- 3.5 Transforming with templates 99
- 3.6 Understanding Camel type converters 100
- 3.7 Summary and best practices 105

Part 1

First steps

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

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

Meeting Camel

1

This chapter covers

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

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

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

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

Are you ready? Let's meet Camel.

1.1 **Introducing Camel**

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

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

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

1.1.1 **What is Camel?**

At the core of the Camel framework is a routing engine—or more precisely, a routing-engine builder. It allows you to define your own routing rules, decide from which sources to accept messages, and determine how to process and send those messages to other destinations. Camel uses an integration language that allows you to define complex routing rules, akin to business processes. As shown in Figure 1.1, Camel forms the glue between disparate systems.

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



Figure 1.1 Camel is the glue between disparate systems.

Camel offers higher-level abstractions that allow you to interact with various systems by using the same API regardless of the protocol or data type the systems are using. Components in Camel provide specific implementations of the API that target different protocols and data types. Out of the box, Camel comes with support for more than 280 protocols and data types. Its extensible and modular architecture allows you to

implement and seamlessly plug in support for your own protocols, proprietary or not. These architectural choices eliminate the need for unnecessary conversions and make Camel not only faster but also lean. As a result, it's suitable for embedding into other projects that require Camel's rich processing capabilities. Other open source projects, such as Apache ServiceMix, Karaf, and ActiveMQ, already use Camel as a way to carry out integration.

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

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

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

1.1.2 Why use Camel?

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

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

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

ROUTING AND MEDIATION ENGINE

The core feature of Camel is its routing and mediation engine. A *routing engine* selectively moves a message around, based on the route's configuration. In Camel's case,

routes are configured with a combination of enterprise integration patterns and a domain-specific language, both of which we'll describe next.

EXTENSIVE COMPONENT LIBRARY

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



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

ENTERPRISE INTEGRATION PATTERNS

Although integration problems are diverse, Gregor Hohpe and Bobby Woolf noticed that many problems, and their solutions are quite similar. They cataloged them in their book *Enterprise Integration Patterns* (Addison-Wesley, 2003), a must-read for any integration professional (www.enterpriseintegrationpatterns.com). If you haven't read it, we encourage you to do so. At the very least, it'll help you understand Camel concepts faster and easier.

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

DOMAIN-SPECIFIC LANGUAGE

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

- Java DSL

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

- XML DSL

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

These examples are real code, and they show how easily you can route files from a folder to a Java Message Service (JMS) queue. Because there's a real programming language underneath, you can use the existing tooling support, such as code completion and compiler error detection, as illustrated in figure 1.3.

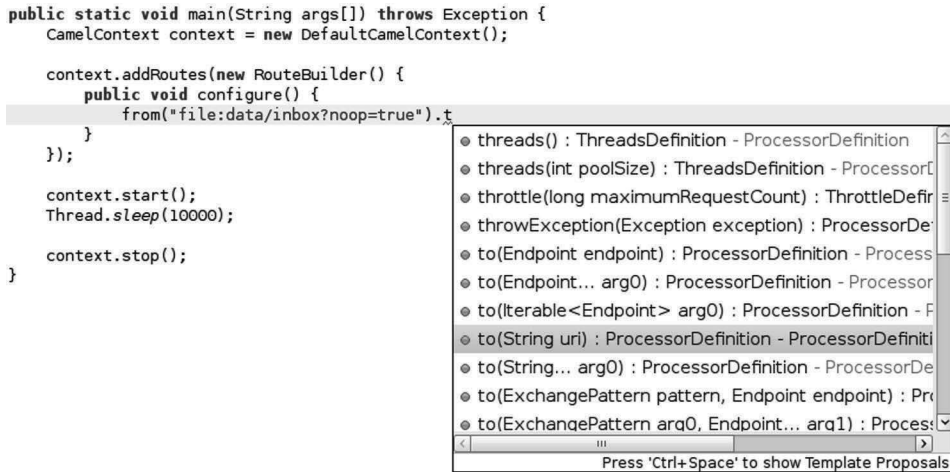


Figure 1.3 Camel DSLs use real programming languages such as Java, so you can use existing tooling support.

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

PAYLOAD-AGNOSTIC ROUTER

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

MODULAR AND PLUGGABLE ARCHITECTURE

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

POJO MODEL

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

EASY CONFIGURATION

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

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

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

AUTOMATIC TYPE CONVERTERS

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

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

LIGHTWEIGHT CORE IDEAL FOR MICROSERVICES

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

CLOUD READY

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

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

TEST KIT

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

VIBRANT COMMUNITY

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

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

1.2 **Getting started**

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

Let's first get the Camel distribution.

1.2.1 **Getting Camel**

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

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

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

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

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

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

- *NOTICE.txt*—Contains copyright information about the third-party dependencies included in the Camel distribution.
- *README.txt*—Contains a short intro to Camel and a list of helpful links to get new users up and running.

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

1.2.2 Your first Camel ride

So far, we've shown you how to get a Camel distribution and offered a peek at what's inside. At this point, feel free to explore the distribution; all examples have instructions to help you figure them out.

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

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

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

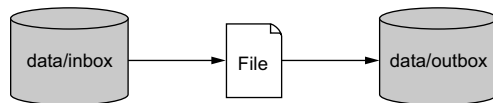


Figure 1.4 Files are routed from the `data/inbox` directory to the `data/outbox` directory.

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

Listing 1.1 Routing files from one folder to another in plain Java

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

public class FileCopier {

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



```

        outboxDirectory.getPath()
        + File.separator
        + source.getName());
    copyFile(source, dest);
    }
}

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

```

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

Integration tasks like these have been done thousands of times before; you shouldn’t ever need to code something like this by hand. Let’s not reinvent the wheel here. Let’s see what a polling solution looks like if you use an integration framework such as Apache Camel.

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

```

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

public class FileCopierWithCamel {

    public static void main(String args[]) throws Exception {
        CamelContext context = new DefaultCamelContext();
        context.addRoutes(new RouteBuilder() {
            public void configure() {
                from("file:data/inbox?noop=true")
                    .to("file:data/outbox");
            }
        });
        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 focus on in listing 1.2 is the *route* ❶.

Routes in Camel are defined in such a way that they flow when read. This route can be read like this: consume messages from file location `data/inbox` with the `noop` option set, and send to file location `data/outbox`. The `noop` option tells Camel to leave the source file as is. If you didn't use this option, the file would be moved. Most people who've never seen Camel before will be able to understand what this route does. You may also want to note that, excluding the boilerplate code, you created a file-polling route in just two lines of Java code ❶.

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

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

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

The POM is shown in the following listing.

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

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.camelinaction</groupId> ← ❶ Parent POM
    <artifactId>chapter1</artifactId>
    <version>2.0.0</version>
  </parent>

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

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

❷ Camel's core library

```

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
</dependency>
</dependencies>
</project>

```

| **Logging support**

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

The Maven POM in listing 1.3 is probably one of the shortest POMs you'll ever see—almost everything uses the defaults provided by Maven. Besides those defaults, some settings are configured in the parent POM **1**. Probably the most important section to point out here is the dependency on the Camel library **2**. This dependency element tells Maven to do the following:

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

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

```
mvn compile exec:java
```

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

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

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

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

1.3 Camel's message model

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

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

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

1.3.1 Message

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

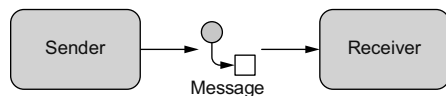


Figure 1.5 Messages are entities used to send data from one system to another.

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

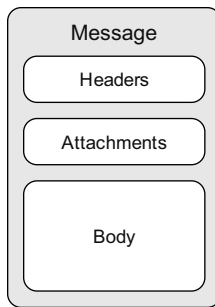


Figure 1.6 A message can contain headers, attachments, and a body.

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

HEADERS AND ATTACHMENTS

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

Body

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

FAULT FLAG

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

During routing, messages are contained in an exchange.

1.3.2 Exchange

An *exchange* in Camel is the message's container during routing. An exchange also provides support for the various types of interactions between systems, also known as *message exchange patterns (MEPs)*. MEPs are used to differentiate between one-way and

request-response messaging styles. The Camel exchange holds a pattern property that can be either of the following:

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

Figure 1.7 illustrates the contents of an exchange in Camel.

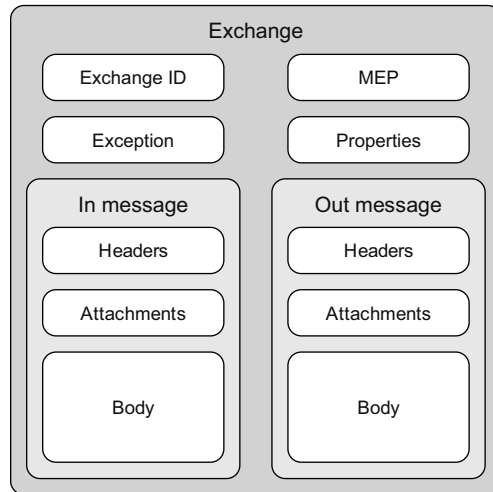


Figure 1.7 A Camel exchange has an ID, MEP, exception, and properties. It also has an *In* message to store the incoming message, and an *Out* message to store the reply.

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

- *Exchange ID*—A unique ID that identifies the exchange. Camel automatically generates the unique ID.
- *MEP*—A pattern that denotes whether you're using the *InOnly* or *InOut* messaging style. When the pattern is *InOnly*, the exchange contains an in message. For *InOut*, an out message also exists that contains the reply message for the caller.
- *Exception*—If an error occurs at any time during routing, an `Exception` will be set in the exception field.
- *Properties*—Similar to message headers, but they last for the duration of the entire exchange. Properties are used to contain global-level information, whereas message headers are specific to a particular message. Camel itself adds various properties to the exchange during routing. You, as a developer, can store and retrieve properties at any point during the lifetime of an exchange.
- *In message*—This is the input message, which is mandatory. The in message contains the request message.
- *Out message*—This is an optional message that exists only if the MEP is *InOut*. The out message contains the reply message.

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

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

1.4 Camel's architecture

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

1.4.1 Architecture from 10,000 feet

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

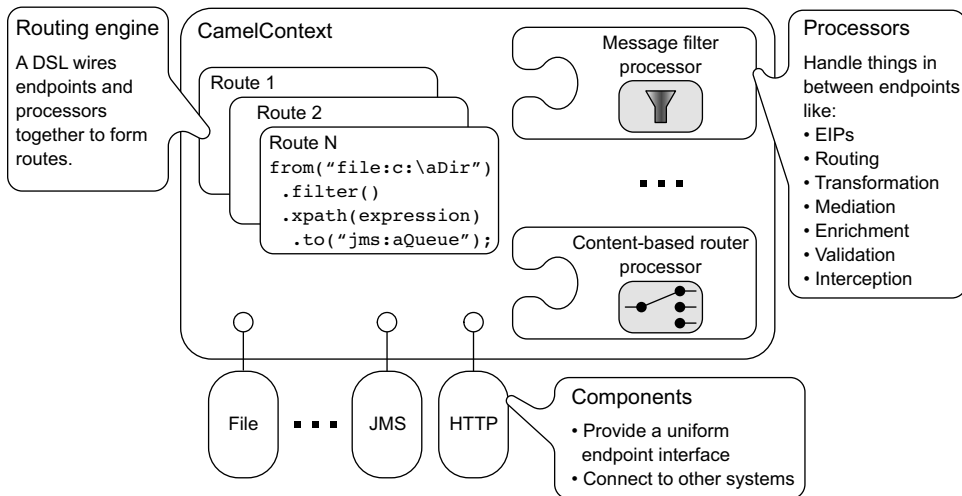


Figure 1.8 At a high level, Camel is composed of routes, processors, and components. All of these are contained within CamelContext.

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

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

1.4.2 Camel concepts

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

CAMELCONTEXT

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

Figure 1.9 shows the most notable services that CamelContext keeps together.

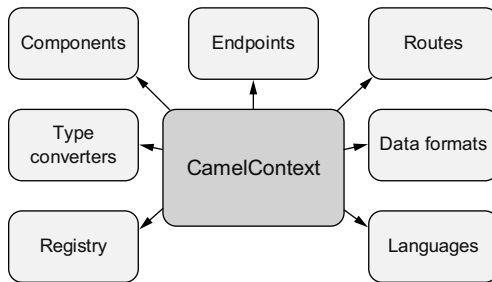


Figure 1.9 CamelContext provides access to many useful services, the most notable being components, type converters, a registry, endpoints, routes, data formats, and languages.

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

Table 1.1 The services that CamelContext provides

Service	Description
Components	Contains the components used. Camel is capable of loading components on the fly either by autodiscovery on the classpath or when a new bundle is activated in an OSGi container. Chapter 6 covers components in more detail.
Endpoints	Contains the endpoints that have been used.
Routes	Contains the routes that have been added. Chapter 2 covers routes.
Type converters	Contains the loaded type converters. Camel has a mechanism that allows you to manually or automatically convert from one type to another. Type converters are covered in chapter 3.
Data formats	Contains the loaded data formats. Data formats are covered in chapter 3.
Registry	Contains a registry that allows you to look up beans. We cover registries in chapter 4.
Languages	Contains the loaded languages. Camel allows you to use many languages to create expressions. You'll get a glimpse of the XPath language in just a moment. A complete reference to Camel's own Simple expression language is available in appendix A.

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

ROUTING ENGINE

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

ROUTES

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

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

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

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

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

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

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

DOMAIN-SPECIFIC LANGUAGE

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

Consider this example:

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

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

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

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

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

PROCESSOR

The *processor* is a core Camel concept that represents a node capable of using, creating, or modifying an incoming exchange. During routing, exchanges flow from one processor to another; as such, you can think of a route as a graph having specialized processors as the nodes, and lines that connect the output of one processor to the input of another. Processors could be implementations of EIPs, producers for specific components, or your own custom creation. Figure 1.10 shows the flow between processors.

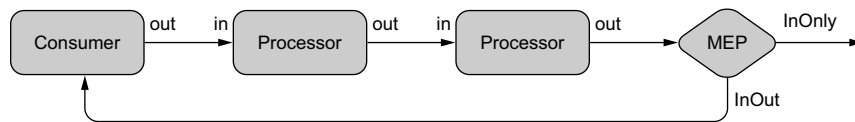


Figure 1.10 Flow of an exchange through a route. Notice that the MEP determines whether a reply will be sent back to the caller of the route.

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

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

systems and bring them into the route. Producers, on the other hand, send (produce) messages to external systems.

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

COMPONENT

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

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

ENDPOINT

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

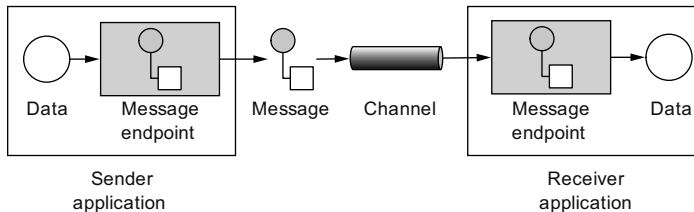


Figure 1.11 An endpoint acts as a neutral interface allowing systems to integrate.

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

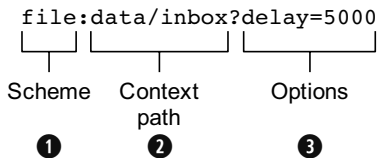


Figure 1.12 Endpoint URIs are divided into three parts: a scheme, a context path, and options.

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

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

PRODUCER

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

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

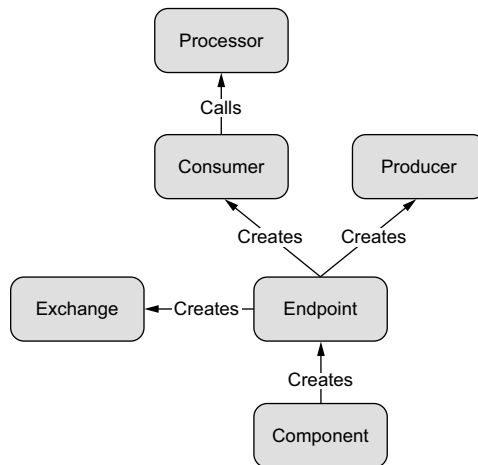


Figure 1.13 How endpoints work with producers, consumers, and an exchange

CONSUMER

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

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

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

EVENT-DRIVEN CONSUMER

The most familiar consumer is probably the *event-driven consumer*, which is illustrated in figure 1.14.

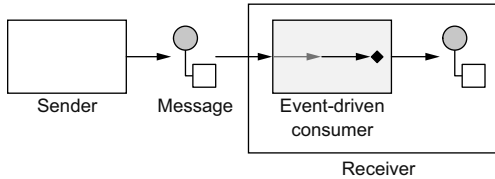


Figure 1.14 An event-driven consumer remains idle until a message arrives, at which point it wakes up and consumes the message.

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

POLLING CONSUMER

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

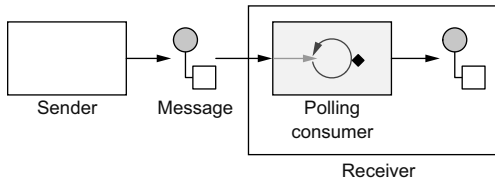


Figure 1.15 A polling consumer actively checks for new messages.

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

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

1.5 Your first Camel ride, revisited

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

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

Listing 1.4 Routing files from one folder to another with Camel

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

public class FileCopierWithCamel {

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

① Java DSL route

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

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

1.6 Summary

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

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

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

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

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

Routing with Camel



This chapter covers

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

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

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

In the context of enterprise messaging systems, routing is the process by which a message is taken from an input queue and, based on a set of conditions, sent to

one of several output queues, as shown in figure 2.1. The input and output queues are unaware of the conditions in between them. The conditional logic is decoupled from the message consumer and producer.

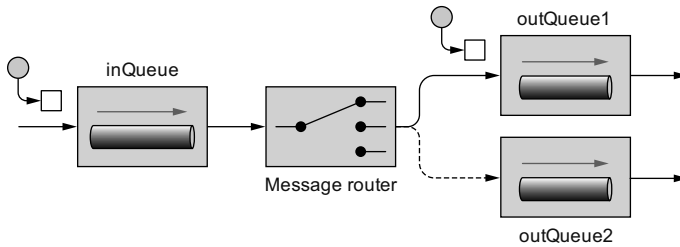


Figure 2.1 A message router consumes messages from an input channel and, depending on a set of conditions, sends the message to one of a set of output channels.

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

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

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

2.1 *Introducing Rider Auto Parts*

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

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

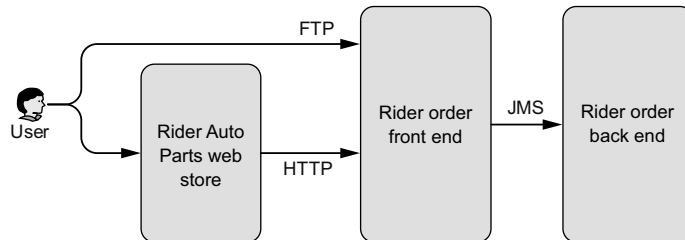


Figure 2.2 A customer has two ways of submitting orders to the Rider Auto Parts order-handling system: either by uploading the raw order file to an FTP server or by submitting an order through the Rider Auto Parts web store. All orders are eventually sent via JMS for processing at Rider Auto Parts.

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

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

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

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

2.2 Understanding endpoints

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

2.2.1 Consuming from an FTP endpoint

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

Take your first Rider Auto Parts assignment, for example. To download new orders from the FTP server, you need to do the following:

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

As shown in figure 2.3, you can easily configure Camel to do this by using URI notation.

```
ftp://rider.com/orders?username=rider&password=secret
```

The diagram shows the URI `ftp://rider.com/orders?username=rider&password=secret` with brackets underneath. The first bracket under `ftp://` is labeled "Scheme". The second bracket under `rider.com/orders` is labeled "Context path". The third bracket under `?username=rider&password=secret` is labeled "Options".

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

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

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

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

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

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

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

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

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

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

2.2.2 Sending to a JMS endpoint

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

WHAT IS JMS?

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

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

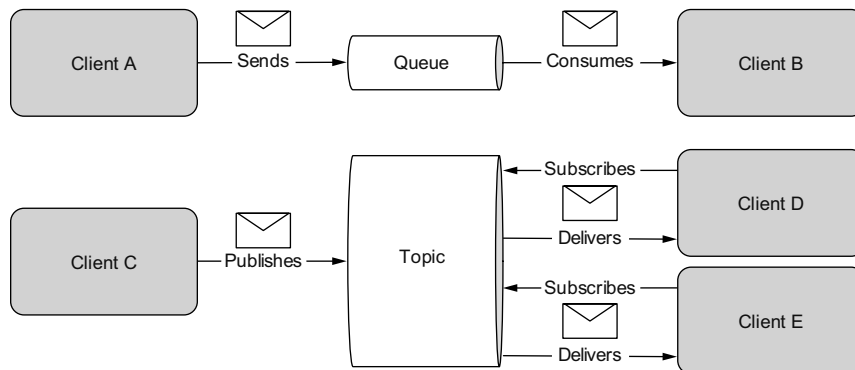


Figure 2.4 There are two types of JMS destinations: queues and topics. The *queue* is a point-to-point channel; each message has only one recipient. A *topic* delivers a copy of the message to all clients that have subscribed to receive it.

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

HOW TO CONFIGURE CAMEL TO USE A JMS PROVIDER

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

Apache ActiveMQ is one of the most popular open source JMS providers, and it's the primary JMS broker that the Camel team uses to test the JMS component. As such, we'll be using it to demonstrate JMS concepts within the book. For more information on Apache ActiveMQ, we recommend *ActiveMQ in Action* by Bruce Snyder et al. (Manning, 2011).

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

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

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

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

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

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

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

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

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

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

USING URIs TO SPECIFY THE DESTINATION

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

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

```
jms:queue:incomingOrders
```

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

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

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

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

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

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

2.3 Creating routes in Java

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

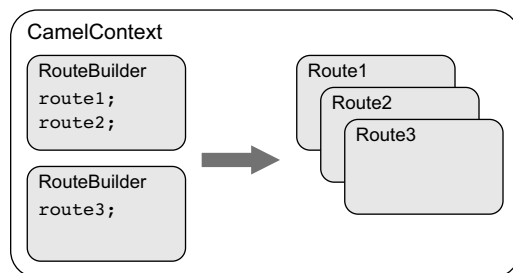


Figure 2.5 RouteBuilders are used to create routes in Camel. Each RouteBuilder can create multiple routes.

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

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

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

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

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

2.3.1 Using `RouteBuilder`

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

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

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

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

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

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

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

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

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

NOTE Eclipse is a popular open source IDE that you can find at <http://eclipse.org>. During the book's development, Jon used Eclipse and Claus used IDEA.

You can certainly use other Java IDEs as well, or even no IDE, but using an IDE does make Camel development a lot easier. Feel free to skip to the next section if you don't want to see the IDE-related setup. In chapter 19, you will see some additional Camel tooling you can install in Eclipse or IDEA that makes Camel development even better.

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

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

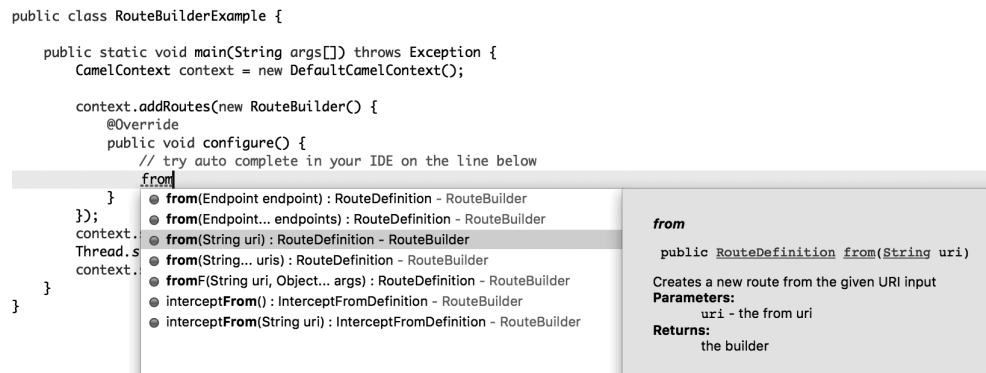


Figure 2.6 Use autocomplete to start your route. All routes start with a `from` method.

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

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

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

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

2.3.2 Using the Java DSL

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

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

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

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

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

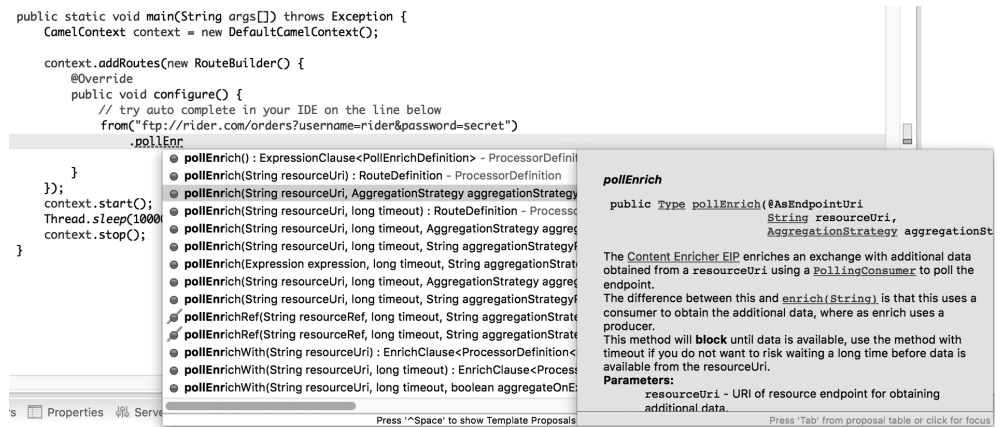


Figure 2.7 After the `from` method, use your IDE’s autocomplete feature to get a list of EIPs (such as `Enricher` and `Recipient List`) and other useful integration functions.

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

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

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

public class FtpToJMSExample {
    public static void main(String args[]) throws Exception {
        CamelContext context = new DefaultCamelContext();
        ConnectionFactory connectionFactory =
            new ActiveMQConnectionFactory("vm://localhost");
        context.addComponent("jms",
            JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));

        context.addRoutes(new RouteBuilder() {
            public void configure() {
                from("ftp://rider.com/orders"
                    + "?username=rider&password=secret")
                    .to("jms:incomingOrders");
            }
        });

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

❶ 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 useful only for demonstrating the Java DSL constructs. For runnable FTP examples, see chapter 6.

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

The flow of messages in this simple route can be viewed as a basic pipeline: the output of the consumer is fed into the producer as input. This is depicted in figure 2.8.

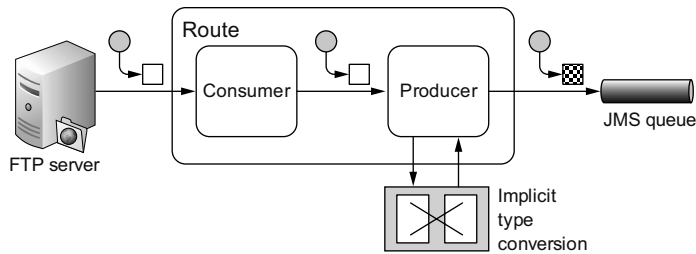


Figure 2.8 The payload conversion from file to JMS message is done automatically.

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

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

ADDING A PROCESSOR

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

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

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

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

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

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

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

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

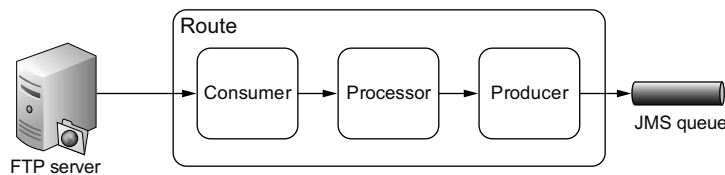


Figure 2.9 With a processor in the mix, the output of the FTP consumer is now fed into the processor, and then the output of the processor is fed into the JMS producer.

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

2.4 Defining routes in XML

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

At the time of this writing, you can write XML routes in two Inversion of Control (IoC) Java containers: Spring and OSGi Blueprint. An IoC framework allows you to "wire" beans together to form applications. This wiring is typically done through an XML configuration file. This section gives you a quick introduction to creating applications with Spring so the IoC concept becomes clear. We'll then show you how Camel uses Spring to form a replacement or complementary solution to the Java DSL.

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

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

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

2.4.1 **Bean injection and Spring**

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

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

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

This interface is implemented by the following classes:

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

You can now create a greeter application as follows:

```
public class GreetMeBean {
    private Greeter greeter;

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

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

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

This XML file instructs Spring to do the following:

- 1 Create an instance of `EnglishGreeter` and name the bean `myGreeter`
- 2 Create an instance of `GreetMeBean` and name the bean `greetMeBean`
- 3 Set the reference of the `greeter` property of the `GreetMeBean` to the bean named `myGreeter`

This configuring of beans is called *wiring*.

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

Here's the final version of `GreetMeBean`:

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

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

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

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

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

```
Hello janstey
```

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

```
Davs janstey
```

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

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

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

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

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

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

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

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

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

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

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

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

```

    </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`. To reproduce the Java DSL example in listing 2.1, you have to factor out the anonymous `RouteBuilder` into its own named class. The `FtpToJMSRoute` class looks like this:

```

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

```

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

2.4.2 The XML DSL

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

Let's continue with the Rider Auto Parts example shown in listing 2.2, but this time you'll specify the routing rules defined in `RouteBuilder` purely in XML. The Spring XML in the following listing does this.

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

```

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

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

```



```

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

```

In this listing, under the `camelContext` element, you replace `routeBuilder` with the `route` element. Within the `route` element, you specify the route by using elements with names similar to ones used inside the Java DSL `RouteBuilder`. Notice that we had to modify the FTP endpoint URI to ensure that it's valid XML. The ampersand character (&) used to define extra URI options is a reserved character in XML, so you have to escape it by using `&`. With this small change, this listing is functionally equivalent to the Java DSL version in listing 2.1 and the Spring plus Java DSL combo in listing 2.2.

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

```

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

```

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

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

ADDING A PROCESSOR

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

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

```

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

```

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

```

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

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

```

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

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

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

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

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

USING MULTIPLE ROUTES

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

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

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

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

CHOOSING WHICH DSL TO USE

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

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

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

2.4.3 Using Camel and Spring

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

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

FINDING ROUTE BUILDERS

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

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

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

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

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

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

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

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

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

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

```
@Component
public class FtpToJMSRoute extends RouteBuilder {
  public void configure() {
```

```

    from("ftp://rider.com" +
        "/orders?username=rider&password=secret")
        .to("jms:incomingOrders");
    }
}

```

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

```

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

```

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

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

CONFIGURING COMPONENTS AND ENDPOINTS

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

```

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

```

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

```

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

```

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

```

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <endpoint id="ridersFtp"
    uri="ftp://rider.com/orders?username=rider&password=secret"/>
  <route>
    <from ref="ridersFtp"/>

```

```

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

```

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

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

```

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

```

IMPORTING CONFIGURATION AND ROUTES

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

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

```

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

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

```

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

```

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

```

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

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

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

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

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

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

SETTING ADVANCED SPRING CONFIGURATION OPTIONS

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

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

2.5 Endpoints revisited

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

2.5.1 Sending to dynamic endpoints

Endpoint URIs like the JMS one shown in section 2.2.2 are evaluated just once when Camel starts up, so they're static entities in Camel. This is fine for most scenarios, when you know ahead of time what the destination names will be called. But what if you need to determine these names at runtime? Static endpoint URIs provided to the to method

will be of no use in this case, because they're evaluated only once at startup. Camel provides an additional DSL method for this: `toD`.

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

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

And in the XML DSL:

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

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

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

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

2.5.2 *Using property placeholders in endpoint URIs*

Rather than having hardcoded endpoint URIs, Camel allows you to use property placeholders in the URIs to replace the dynamic parts. By *dynamic*, we mean that values will be replaced when Camel starts up, not on every new message, as in the case of `toD` described in the previous section.

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

USING THE PROPERTIES COMPONENT

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

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

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

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

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

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

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

```
myDest=incomingOrders
```

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

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

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

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

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

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

USING PROPERTY PLACEHOLDERS IN THE XML DSL

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

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

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

```
myDest=incomingOrders
```


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

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

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

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

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

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

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

USING SPRING PROPERTY PLACEHOLDERS

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

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

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

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

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

```

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

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

```

Unfortunately, the Spring Framework doesn't support using placeholders directly in endpoint URIs in the route, so you must define endpoints that include those placeholders by using the `<endpoint>` tag. The following code snippet shows how this is done:

```

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

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

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

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

```

1 Loads properties from external file

2 Defines endpoint using Spring 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 ❶ that will load the properties file from the classpath.

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

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

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

```
myDest=incomingOrders
```

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

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

The Camel Properties component vs. Spring property placeholders

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

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

2.5.3 Using raw values in endpoint URIs

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

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

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

2.5.4 Referencing registry beans in endpoint URIs

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

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

Add it to the registry:

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

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

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

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

2.6 Routing and EIPs

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

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

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

2.6.1 Using a content-based router

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

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

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

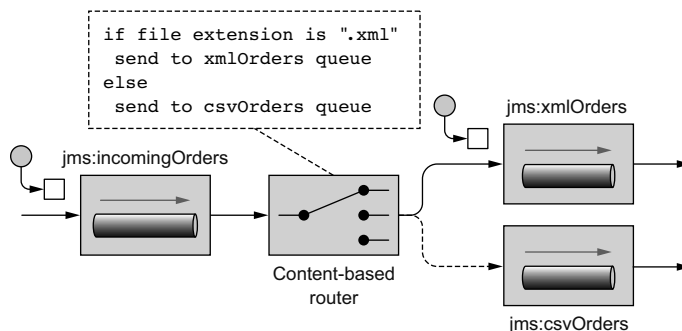


Figure 2.10 The CBR routes messages based on their content. In this case, the filename extension (as a message header) is used to determine which queue to route to.

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

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

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

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

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

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

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

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

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

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

The completed CBR is shown in the following listing.

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

```
return new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        // load file orders from src/data into the JMS queue
        from("file:src/data?noop=true").to("jms:incomingOrders");
    }
}
```

```

from("jms:incomingOrders")
    .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 consumes two order files in the `chapter2/cbr/src/data` directory and outputs the following:

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

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

The Simple language

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

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

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

```

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

```

Here you use the `ends with` operator to check the end of the string returned from the `${header.CamelFileName}` dynamic simple expression. The Simple language is so useful for Camel applications that we devote appendix A to cover it fully.

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

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

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

```

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

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

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

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

```

❶ Simple expression used instead of Java-based predicate

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=SpringOrderRouterTest
```

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

USING THE OTHERWISE CLAUSE

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

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

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

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

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

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

The equivalent route in XML DSL is as follows:

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

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

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

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

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


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

ROUTING AFTER A CONTENT-BASED ROUTER

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

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

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

```
from("jms:incomingOrders")
  .choice()
    .when(header("CamelFileName").endsWith(".xml"))
      .to("jms:xmlOrders")
    .when(header("CamelFileName").regex("^.*(csv|cs1)$"))
      .to("jms:csvOrders")
    .otherwise()
      .to("jms:badOrders")
  .end()
  .to("jms:continuedProcessing");
```

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

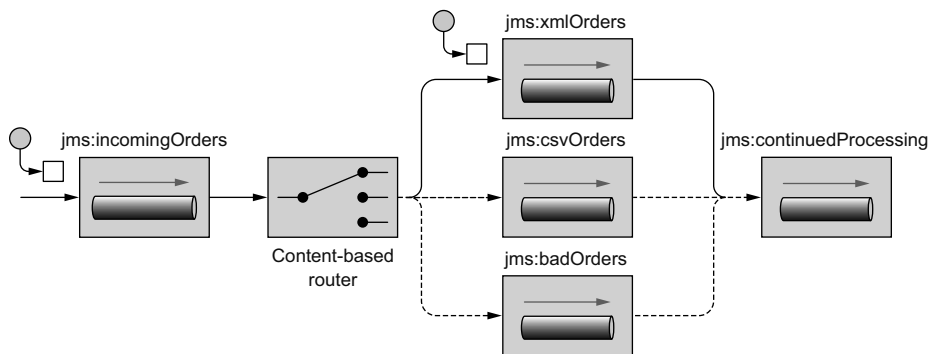


Figure 2.11 By using the end method, you can route messages to a destination after the CBR.

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

```
from("jms:incomingOrders")
  .choice()
    .when(header("CamelFileName").endsWith(".xml"))
      .to("jms:xmlOrders")
    .when(header("CamelFileName").regex("^.*(csv|cs1)$"))
      .stop();
```

```

        .to("jms:csvOrders")
    .otherwise()
        .to("jms:badOrders").stop()
    .end()
    .to("jms:continuedProcessing");

```

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

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

```

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

```

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

2.6.2 Using message filters

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

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

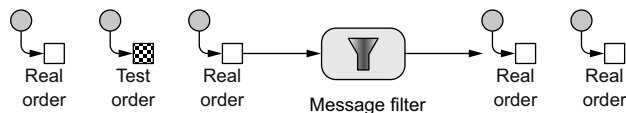


Figure 2.12 A message filter allows you to filter out uninteresting messages based on a certain condition. In this case, test messages are filtered out.

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

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

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

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

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

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

This outputs the following on the command line:

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

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

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

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

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

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

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

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

2.6.3 *Using multicasting*

Often in enterprise applications you'll need to send a copy of a message to several destinations for processing. When the list of destinations is known ahead of time and is static, you can add an element to the route that will consume messages from a source

endpoint and then send the message out to a list of destinations. Borrowing terminology from computer networking, we call this the Multicast EIP.

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

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

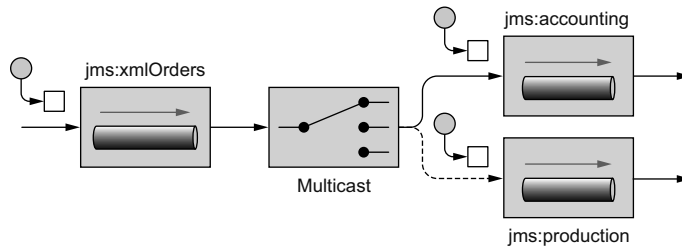


Figure 2.13 A multicast sends a message to numerous specified recipients.

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

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

The equivalent route in XML DSL is as follows:

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

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

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

You should see the following output on the command line:

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

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

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

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

USING PARALLEL MULTICASTING

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

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

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

The equivalent route in XML DSL is as follows:

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

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

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

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

STOPPING THE MULTICAST ON EXCEPTION

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

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

```
from("jms:xmlOrders")
    .multicast()
      .stopOnException()
      .to("direct:accounting", "direct:production")
    .end()
    .to("mock:end");

from("direct:accounting")
    .throwException(Exception.class, "I failed!")
    .log("Accounting received order: ${header.CamelFileName}")
    .to("mock:accounting");

from("direct:production")
    .log("Production received order: ${header.CamelFileName}")
    .to("mock:production");
```

To handle the exception coming back from this route, you'll need to use Camel's error-handling facilities, which are described in detail in chapter 11.

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

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

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

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

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

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

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

2.6.4 Using recipient lists

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

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

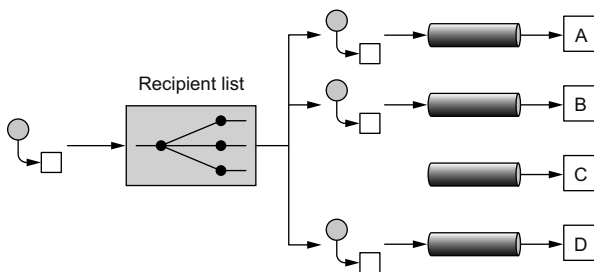


Figure 2.14 A recipient list inspects the incoming message and determines a list of recipients based on the content of the message. In this case, the message contains a list of destinations such as A, B, D. So Camel sends the message to only the A, B, and D destinations. The next message could contain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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

RECIPIENT LIST ANNOTATION

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

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

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

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

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

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

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

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

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

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

One nice thing about implementing the recipient list this way is that it's entirely separated from the route, which makes it a bit easier to read. You also have access to Camel's bean-binding annotations, which allow you to extract data from the message by using expressions, so you don't have to manually explore the exchange. This example uses the `@XPath` bean-binding annotation to grab the customer attribute of the order element in the body. We cover these annotations in chapter 4, which is all about using beans. To run this example, go to the `chapter2/recipientlist` directory in the book's source code and run the command for either the Java or XML DSL case:

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

This outputs the following on the command line:

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

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

- message1.xml

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

- message2.xml

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

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

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

2.6.5 Using the wireTap method

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

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

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

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

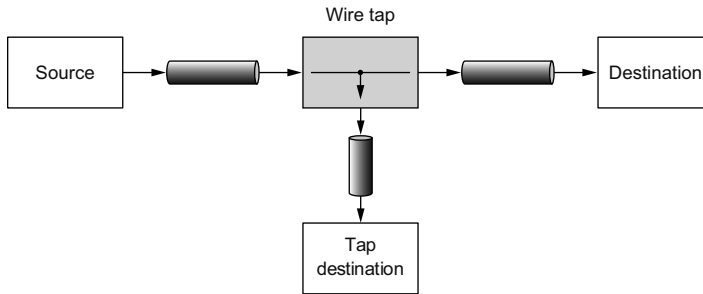


Figure 2.15 A wire tap is a fixed recipient list that sends a copy of a message traveling from a source to a destination to a secondary destination.

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

```

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

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

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

```

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

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

```

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

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

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

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

2.7 Summary and best practices

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

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

- *Routing occurs in many aspects of everyday life*—Whether you're surfing the internet, doing online banking, or booking a flight or hotel room, messages are being routed behind the scenes via some sort of router.
- *Use Apache Camel for routing messages*—Camel is primarily a message router that allows you to route messages from and to a variety of transports and APIs.
- *Camel's DSLs are used to define routing rules*—The Java DSL allows you to write in the popular Java language, which gives you autocompletion of terms in most IDEs. It also allows you to use the full power of the Java language when writing routes. It's considered the main DSL in Camel. The XML DSL allows you to write routing rules without any Java code at all.
- *The Java DSL and Spring CamelContext are a powerful combination*—Section 2.4.3 described our favorite way to write Camel applications, which is to boot up CamelContext in Spring and write routing rules in Java DSL RouteBuilders. This gives you the best of both: the most expressive DSL that Camel has in the Java DSL, and a more feature-rich and standard container in the Spring CamelContext.
- *Use enterprise integration patterns (EIPs) to solve integration and routing problems*—EIPs are like design patterns from object-oriented programming, but for the enterprise integration world.
- *Use Camel's built-in EIP implementations rather than creating your own*—Camel implements most EIPs as easy-to-use DSL terms, which allows you to focus on the business problem rather than the integration architecture.

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

Part 2

Core Camel

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

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

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

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

Transforming data with Camel

This chapter covers

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

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

Just like the real world, where people speak different languages, the IT world speaks different protocols. Software engineers regularly need to act as mediators between various protocols when IT systems must be integrated. To address this, the data models used by the protocols must be transformed from one form to another,

adapting to whatever protocol the receiver understands. Mediation and data transformation are key features in any integration kit, including Camel.

In this chapter, you'll learn all about how Camel can help you with your data transformation challenges. We'll start with a brief overview of data transformation in Camel and then look at transforming data into any custom format you may have. Next we'll look at Camel components that are specialized for transforming XML data and other well-known data formats. We end the chapter by looking into Camel's type-converter mechanism, which supports, implicitly and explicitly, type conversion.

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

3.1 **Data transformation overview**

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

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

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

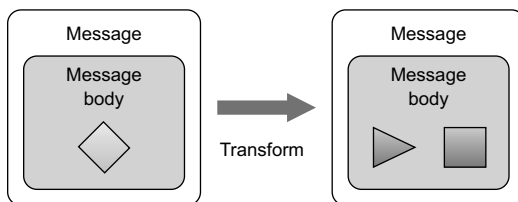


Figure 3.1 Camel offers many features for transforming data from one form to another.

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

Table 3.1 Six ways data transformation typically takes place in Camel

Transformation	Description
Data transformation using EIPs and Java	You can explicitly enforce transformation in the route by using the Message Translator or the Content Enricher EIPs. This gives you the power to do data mapping by using regular Java code. We cover this in section 3.2.
Data transformation using components	Camel provides a range of components for transformation, such as the XSLT component for XML transformation. We dive into this in section 3.3.
Data transformation using data formats	Data formats are Camel transformers that come in pairs to transform data back and forth between well-known formats. Section 3.4 covers this topic.
Data transformation using templates	Camel provides a range of components for transforming by using templates, such as Apache Velocity. We'll look at this in section 3.5.
Data type transformation using Camel's type-converter mechanism	Camel has an elaborate type-converter mechanism that activates on demand. This is convenient when you need to convert from common types such as <code>java.lang.Integer</code> to <code>java.lang.String</code> or even from <code>java.io.File</code> to <code>java.lang.String</code> . Section 3.6 covers type converters.
Message transformation in component adapters	Camel's many components adapt to various commonly used protocols and, as such, need to be able to transform messages as they travel to and from those protocols. Often these components use a combination of custom data transformations and type converters. This happens seamlessly, and only component writers need to worry about it. Chapter 8 covers writing custom components.

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

3.2 Transforming data by using EIPs and Java

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

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

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

3.2.1 Using the Message Translator EIP

The Message Translator EIP is illustrated in figure 3.2.

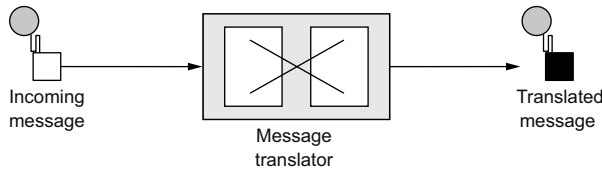


Figure 3.2 In the Message Translator EIP, an incoming message goes through a translator and comes out as a translated message.

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

NOTE The Gang of Four book is *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994). See the “Design Patterns” Wikipedia article for more information: [http://en.wikipedia.org/wiki/Design_Patterns_\(book\)](http://en.wikipedia.org/wiki/Design_Patterns_(book)).

Camel provides three ways of using this pattern:

- Using Processor
- Using Java beans
- Using `<transform>`

We'll look at them each in turn.

TRANSFORMING USING PROCESSOR

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

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

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

Let's look at an example. At Rider Auto Parts, you've been asked to generate daily reports of newly received orders to be outputted to a CSV file. The company uses a custom format for order entries, but to make things easy, they already have an HTTP service that returns a list of orders for whatever date you input. The challenge you face is mapping the returned data from the HTTP service to a CSV format and writing the report to a file.

Because you want to get started on a prototype quickly, you decide to use the Camel Processor, as shown in the following listing.

Listing 3.1 Using Processor to translate from a custom format to a CSV format

```

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

public class OrderToCsvProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        String custom = exchange.getIn()
            .getBody(String.class); ← ❶ Gets custom payload
        String id = custom.substring(0, 10);
        String customerId = custom.substring(10, 20);
        String date = custom.substring(20, 30);
        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());
        }
        exchange.getIn().setBody(csv.toString()); ← ❷ Replaces payload with CSV payload
    }
}

```

❷ Extracts data to local variables

❸ Maps to CSV format

First you grab the custom format payload from the exchange ❶. 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 ❷. 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 ❸. Finally, you replace the custom payload with your new CSV payload ❹.

You can use `OrderToCsvProcessor` from listing 3.1 in a Camel route as follows:

```

from("quartz2://report?cron=0+0+6+*+*+*?")
    .to("http://riders.com/orders/cmd=received&date=yesterday")
    .process(new OrderToCsvProcessor())
    .to("file://riders/orders?fileName=report-${header.Date}.csv");

```

The preceding route uses Quartz to schedule a job to run once a day at 6 a.m. It then invokes the HTTP service to retrieve the orders received yesterday, which are returned in the custom format. Next, it uses `OrderToCsvProcessor` to map from the custom format to CSV format before writing the result to a file.

The equivalent route in XML is as follows:

```

<bean id="csvProcessor" class="camelinaction.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"/>
            </to>
        </route>
    </camelContext>

```

You can try this example yourself; we've provided a little unit test with the book's source code. Go to the `chapter3/transform` directory and run these Maven goals:

```
mvn test -Dtest=OrderToCsvProcessorTest
mvn test -Dtest=SpringOrderToCsvProcessorTest
```

After the test runs, a report file is written in the `target/orders/received` directory.

Using the `getIn` and `getOut` methods on exchanges

The Camel `Exchange` defines two methods for retrieving messages: `getIn` and `getOut`. The `getIn` method returns the incoming message, and the `getOut` method accesses the outbound message.

In two scenarios, the Camel end user will have to decide which method to use:

- A read-only scenario, such as when you're logging the incoming message
- A write scenario, such as when you're transforming the message

In the second scenario, you'd assume `getOut` should be used. That's correct according to theory, but in practice there's a common pitfall when using `getOut`: the incoming message headers and attachments will be lost. This is often not what you want, so you must copy the headers and attachments from the incoming message to the outgoing message, which can be tedious. The alternative is to set the changes directly on the incoming message by using `getIn`, and not to use `getOut` at all. This is the practice we use most often in this book.

Using a processor has one disadvantage: you're required to use the Camel API. In the next section, you'll learn how to avoid this by using a bean.

TRANSFORMING USING BEANS

Using beans is a great practice because it allows you to use any Java code and library you wish. Camel imposes no restrictions whatsoever. Camel can invoke any bean you choose, so you can use existing beans without having to rewrite or recompile them.

The following listing shows using a bean instead of `Processor`.

Listing 3.2 Using a bean to translate from a custom format to CSV format

```
public class OrderToCsvBean {
    public static String map(String custom) {
        String id = custom.substring(0, 10);
        String customerId = custom.substring(10, 20);
        String date = custom.substring(20, 30);
        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());
```

① Extracts data to local variables

```

    }
    return csv.toString(); ← ❷ Returns CSV payload
  }
}

```

The first noticeable difference between listings 3.1 and 3.2 is that listing 3.2 doesn't use any Camel imports. Your bean is totally independent of the Camel API. The next difference is that you can name the method signature in listing 3.2—in this case, it's a static method named `map`.

The method signature defines the contract, which means that the first parameter (`String custom`) is the message body you're going to use for translation. The method returns a string, which means the translated data will be a `String` type. At runtime, Camel binds to this method signature. We won't go into any more details here; chapter 4 covers much more about using beans.

The mapping ❶ is the same as with the processor. At the end, you return the mapping output ❷.

You can use `OrderToCsvBean` in a Camel route as shown here:

```

from("quartz2://report?cron=0+0+6+*+*+*?")
  .to("http://riders.com/orders/cmd=received&date=yesterday")
  .bean(new OrderToCsvBean())
  .to("file://riders/orders?fileName=report-${header.Date}.csv");

```

The equivalent route in XML is as follows:

```

<bean id="csvBean" class="camelinaction.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

```

This generates a test report file in the `target/orders/received` directory.

Another advantage of using beans over processors for mappings is that unit testing is much easier. For example, listing 3.2 doesn't require the use of Camel at all, as opposed to listing 3.1, where you need to create and pass in an `Exchange` instance.

We'll leave the beans for now, because they're covered extensively in the next chapter. But you should keep in mind that beans are useful for doing message transformation.

TRANSFORMING USING THE TRANSFORM METHOD FROM THE JAVA DSL

`transform` is a method in the Java DSL that can be used in Camel routes to transform messages. By allowing the use of expressions, `transform` permits great flexibility, and using expressions directly within the DSL can sometimes save time. Let's look at a little example.

Suppose you need to prepare text for HTML formatting by replacing all line breaks with a `
` tag. You can do this with a built-in Camel expression that searches and replaces using regular expressions:

```
from("direct:start")
  .transform(body().replaceAll("\n", "<br/>"))
  .to("mock:result");
```

What this route does is use the `transform` method to tell Camel that the message should be transformed using an expression. Camel provides the Builder pattern to build compound expressions from individual expressions. This is done by chaining together method calls, which is the essence of the Builder pattern.

NOTE For more information on the Builder pattern, see the Wikipedia article: http://en.wikipedia.org/wiki/Builder_pattern.

In this example, you combine `body` and `replaceAll`. The expression should be read as follows: take the `body` and perform a regular expression that replaces all new lines (`\n`) with `
` tags. Now you've combined two methods that conform to a compound Camel expression.

You can run this example from `chapter3/transform` directly by using the following Maven goal:

```
mvn test -Dtest=TransformTest
```

The Direct component

The example here uses the Direct component (<http://camel.apache.org/direct>) as the input source for the route (`from("direct:start")`). The Direct component provides direct invocation between a producer and a consumer. It allows connectivity only from within Camel, so external systems can't send messages directly to it. This component is used within Camel to do things such as link routes together or for testing.

For more information on the Direct component and other types of in-memory messaging, see chapter 6.

Camel also allows you to use custom expressions. This is useful when you need to be in full control and have Java code at your fingertips. For example, the previous example could've been implemented as follows:

```
from("direct:start")
  .transform(new Expression() {
    public <T> T evaluate(Exchange exchange, Class<T> type) {
      String body = exchange.getIn().getBody(String.class);
      body = body.replaceAll("\n", "<br/>");
      body = "<body>" + body + "</body>";
      return (T) body;
    }
  })
  .to("mock:result");
```

As you can see, this code uses an inlined Camel Expression that allows you to use Java code in its evaluate method. This follows the same principle as the Camel Processor you saw before.

Now let's see how to transform data using the XML DSL.

TRANSFORMING USING <TRANSFORM> FROM THE XML DSL

Using <transform> from the XML DSL is a bit different from the Java DSL because the XML DSL isn't as powerful. In the XML DSL, the Builder pattern expressions aren't available because with XML you don't have a real programming language underneath. What you can do instead is invoke a method on a bean or use scripting languages.

Let's see how this works. The following route uses a method call on a bean as the expression:

```
<bean id="htmlBean" class="camelinaction.HtmlBean"/>
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <transform>
      <method bean="htmlBean" method="toHtml"/>
    </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 straightforward:

```
public class HtmlBean {
    public static String toHtml(String body) {
        body = body.replaceAll("\n", "<br/>");
        body = "<body>" + body + "</body>";
        return body;
    }
}
```

You can also use scripting languages as expressions in Camel. For example, you can use Groovy, MVFLEX Expression Language (MVEL), JavaScript, or Camel's own scripting language, called Simple (explained in appendix A). We won't go into detail on how to use the other scripting languages at this point, but you can use the Simple language to build strings with placeholders. It pretty much speaks for itself—we're sure you'll understand what the following transformation does:

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

You can try the XML DSL transformation examples provided in the book's source code by running the following Maven goals from the chapter3/transform directory:

```
mvn test -Dtest=SpringTransformMethodTest
mvn test -Dtest=SpringTransformScriptTest
```


We're done covering the Message Translator EIP, so let's look at the related Content Enricher EIP.

3.2.2 Using the Content Enricher EIP

The Content Enricher EIP is illustrated in figure 3.3. This pattern documents the scenario in which a message is enriched with data obtained from another resource.

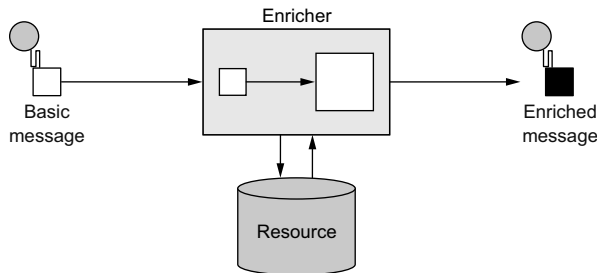


Figure 3.3 In the Content Enricher EIP, an existing message has data added to it from another source.

To help understand this pattern, let's turn back to Rider Auto Parts. It turns out that the data mapping you did in listing 3.1 wasn't sufficient. Orders are also piled up on an FTP server, and your job is to somehow merge this information into the existing report. Figure 3.4 illustrates the scenario.

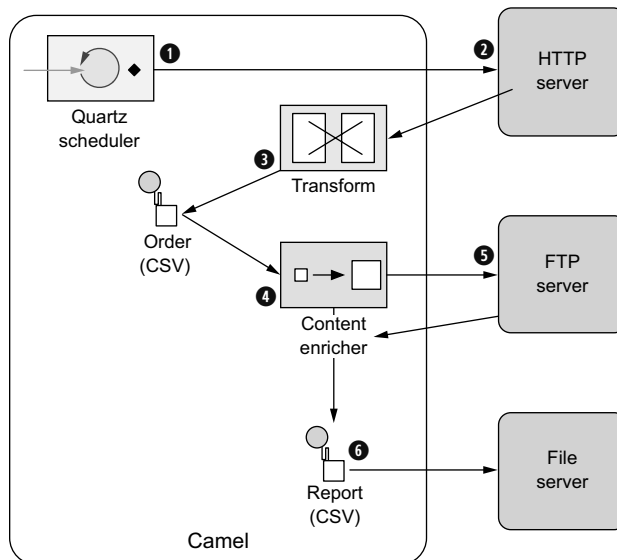


Figure 3.4 An overview of the route that generates the orders report, now with the content enricher pulling in data from an FTP server

A scheduled consumer using Quartz starts the route every day at 6 a.m. ❶. It then pulls data from an HTTP server, which returns orders in a custom format ❷, which is then transformed into CSV format ❸. At this point, you have to perform the additional content enrichment step ❹ with the data obtained from the FTP server ❺. After this, the final report is written to the file server ❻.

Before you dig into the code and see how to implement this, you need to take a step back and look at how the Content Enricher EIP is implemented in Camel. Camel provides two methods in the DSL for implementing the pattern:

- `pollEnrich`—This method merges data retrieved from another source by using a consumer.
- `enrich`—This method merges data retrieved from another source by using a producer.

The difference between `pollEnrich` and `enrich`

The difference between `pollEnrich` and `enrich` is that the former uses a consumer, and the latter uses a producer, to retrieve data from the source. Knowing the difference is important: the file component can be used with both, but using `enrich` will write the message content as a file; using `pollEnrich` will read the file as the source, which is most likely the scenario you'll be facing when enriching with files. The HTTP component works only with `enrich`; it allows you to invoke an external HTTP service and use its reply as the source.

Camel uses the `org.apache.camel.processor.aggregate.AggregationStrategy` interface to merge the result from the source with the original message, as follows:

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

This `aggregate` method is a callback that you must implement. The method has two parameters: the first, named `oldExchange`, contains the original exchange; the second, `newExchange`, is the enriched source. Your task is to enrich the message by using Java code and return the merged result. Let's see this in action.

To solve the problem at Rider Auto Parts, you need to use `pollEnrich` because it's capable of polling a file from an FTP server.

ENRICHING USING `POLLENRICH`

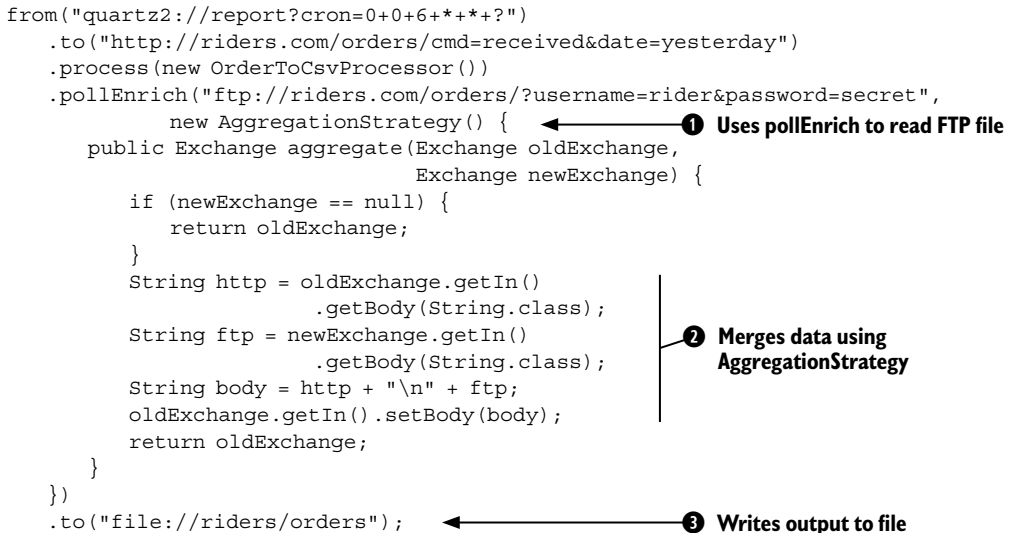
The following listing shows how to use `pollEnrich` to retrieve the additional orders from the remote FTP server and aggregate this data with the existing message by using Camel's `AggregationStrategy`.

Listing 3.3 Using `pollEnrich` to merge additional data with an existing message

```

from("quartz2://report?cron=0+0+6+*+*+*?")
.to("http://riders.com/orders/cmd=received&date=yesterday")
.process(new OrderToCsvProcessor())
.pollEnrich("ftp://riders.com/orders/?username=rider&password=secret",
    new AggregationStrategy() {
        public Exchange aggregate(Exchange oldExchange,
            Exchange newExchange) {
            if (newExchange == null) {
                return oldExchange;
            }
            String http = oldExchange.getIn()
                .getBody(String.class);
            String ftp = newExchange.getIn()
                .getBody(String.class);
            String body = http + "\n" + ftp;
            oldExchange.getIn().setBody(body);
            return oldExchange;
        }
    })
.to("file://riders/orders");

```



The route is triggered by Quartz to run at 6 a.m. every day. You invoke the HTTP service to retrieve the orders and transform them to CSV format by using a processor.

At this point, you need to enrich the existing data with the orders from the remote FTP server. This is done by using `pollEnrich` ❶, which consumes the remote file.

To merge the data, you use `AggregationStrategy` ❷. First, you check whether any data was consumed. If `newExchange` is null, there's no remote file to consume, and you just return the existing data. If there's a remote file, you merge the data by concatenating the existing data with the new data and setting it back on the `oldExchange`. Then, you return the merged data by returning the `oldExchange`. To write the CSV report file, you use the file component ❸.

TIP Both `enrich` and `pollEnrich` can accept dynamic URIs, as discussed in chapter 2, section 2.5.1.

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

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

It's a best practice to either use `timeout = 0` or assign a time-out value when using `pollEnrich` to avoid waiting indefinitely if no message arrives.

Now let's take a quick look at how to use `enrich` with the XML DSL; it's a bit different from using the Java DSL. You use `enrich` when you need to enrich the current message with data from another source using request-reply messaging. A prime example is to enrich the current message with the reply from a web service call. But let's look at another example, using XML to enrich the current message via the TCP transport:

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

← ❶ **Bean implementing
AggregationStrategy**

Here you use the Camel `netty4` component for the TCP transport, configured to use request-reply messaging by using the `sync=true` option. To merge the original message with data from the remote server, `<enrich>` must refer to an `AggregationStrategy`. This is done using the `strategyRef` attribute. As you can see in the example, the `quoteStrategy` being referred to is a bean id ❶, which contains the implementation of the `AggregationStrategy`, where the merging takes place.

You've seen a lot about how to transform data in Camel, using Java code for the transformations. Now let's take a peek into the XML world and look at the XSLT component, which is used for transforming XML messages into another format by using XSLT stylesheets.

3.3 Transforming XML

Camel provides two ways to perform XML transformations:

- *XSLT component*—For transforming an XML payload into another format by using XSLT stylesheets
- *XML marshaling*—For marshaling and unmarshaling objects to and from XML

Both of these are covered in the following subsections.

3.3.1 Transforming XML with XSLT

XSL Transformations (XSLT) is a declarative XML-based language used to transform XML documents into other documents. For example, XSLT can be used to transform XML into HTML for web pages or to transform an XML document into another XML document with a different structure. XSLT is powerful and versatile, but it's also a complex language that takes time and effort to fully understand and master. Think twice before deciding to pick up and use XSLT.

Camel provides the XSLT component as part of `camel-core.jar`, so you don't need any other dependencies. Using the XSLT component is straightforward because it's just another Camel component. The following route shows an example of how to use it; this route is also illustrated in figure 3.5:

```
from("file://rider/inbox")
  .to("xslt://camelinaction/transform.xml")
  .to("jms:queue:transformed")
```

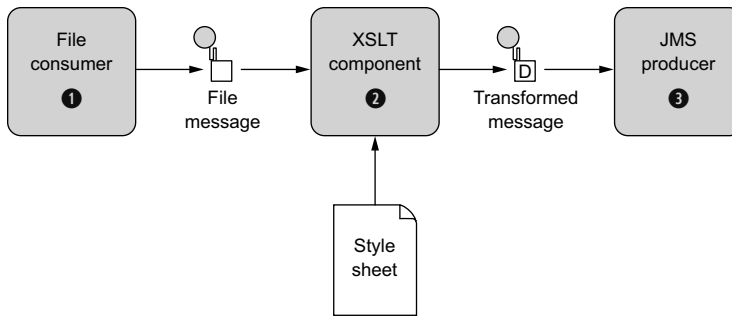


Figure 3.5 A Camel route using an XSLT component to transform an XML document before it's sent to a JMS queue

The file consumer picks up new files and routes them to the XSLT component, which transforms the payload by using the stylesheet. After the transformation, the message is routed to a JMS producer, which sends the message to the JMS queue. Notice in the preceding code how the URI for the XSLT component is defined: `xslt://camelinaction/transform.xml`. 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://camelinaction/transform.xml</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.xml</code>	Loads the resource from a URL.
<code>ref:</code>	<code>xslt://ref:resourceId</code>	Look up the resource from the registry.
<code>bean:</code>	<code>xslt://bean:nameOfBean.methodName</code>	Look up a bean in the registry and call a method which returns the resource.

Let's leave the XSLT world now and take a look at how to do XML-to-object marshaling with Camel.

3.3.2 Transforming XML with object marshaling

Any software engineer who has worked with XML knows that it's a challenge to use the low-level XML API that Java offers. Instead, people often prefer to work with

regular Java objects and use marshaling to transform between Java objects and XML representations.

In Camel, this marshaling process is provided in ready-to-use components known as *data formats*. Section 3.4 covers data formats in full detail, but you'll take a quick look at the XStream and JAXB data formats here as we cover XML transformations using marshaling.

TRANSFORMING USING XSTREAM

XStream is a simple library for serializing objects to XML and back again. To use it, you need camel-xstream.jar on the classpath and the XStream library itself.

Suppose you need to send messages in XML format to a shared JMS queue, which is then used to integrate two systems. The following listing shows how this can be done.

Listing 3.4 Using XStream to transform a message into XML

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <dataFormats>
    <xstream id="myXstream"/>
  </dataFormats>
  <route>
    <from uri="direct:foo"/>
    <marshal ref="myXstream"/>
    <to uri="jms:queue:foo"/>
  </route>
</camelContext>
```

← ❶ Specifies XStream data format

← ❷ Transforms to XML

When using the XML DSL, you can declare the data formats used at the top ❶ 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` ❷, which refers to the `id` from ❶, so Camel knows that the XStream data format is being used.

You can also use the XStream data format directly in the route, which can shorten the syntax a bit, like this:

```
<route>
  <from uri="direct:foo"/>
  <marshal><xstream/></marshal>
  <to uri="jms:queue:foo"/>
</route>
```

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

```
from("direct:foo").marshal().xstream().to("jms:queue:foo");
```

Yes, using XStream is that simple. And the reverse operation, unmarshaling from XML to an object, is just as simple:

```
<route>
  <from uri="jms:queue:foo"/>
  <unmarshal ref="myXstream"/>
  <to uri="direct:handleFoo"/>
</route>
```

You've now seen how easy it is to use XStream with Camel. Let's take a look at using JAXB with Camel.

TRANSFORMING USING JAXB

Java Architecture for XML Binding (JAXB) is a standard specification for XML binding, and it's provided out of the box in the Java runtime. Like XStream, it allows you to serialize objects to XML and back again. It's not as simple, but it does offer more bells and whistles for controlling the XML output. And because it's distributed in Java, you don't need any special JAR files on the classpath.

Unlike XStream, JAXB requires that you do a bit of work to declare the binding between Java objects and the XML form. This is done using annotations. Suppose you define a model bean to represent an order, as shown in listing 3.5, and you want to transform this into XML before sending it to a JMS queue. Then you want to transform it back to the order bean again when consuming from the JMS queue. This can be done as shown in listings 3.5 and 3.6.

Listing 3.5 Annotating a bean with JAXB so it can be transformed to and from XML

```
package camelinaction;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class PurchaseOrder {
    @XmlAttribute
    private String name;
    @XmlAttribute
    private double price;
    @XmlAttribute
    private double amount;
}
```

❶ 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` ❶ as a class-level annotation to indicate that this class is an XML element. Then you define the `@XmlAccessorType` to let JAXB access fields directly. To expose the fields of this model object as XML attributes, you mark them with the `@XmlAttribute` annotation.

Using JAXB, you should be able to marshal a model object into an XML representation like this:

```
<purchaseOrder name="Camel in Action" price="6999" amount="1"/>
```

The following listing shows how you can use JAXB in routes to transform the `PurchaseOrder` object to XML before it's sent to a JMS queue, and then back again from XML to the `PurchaseOrder` object when consuming from the same JMS queue.

Listing 3.6 Using JAXB to serialize objects to and from XML

```

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <dataFormats>
    <jaxb id="jaxb" contextPath="camelinaction"/> ← ❶ Declares JAXB
    </dataFormats>                                data format
  <route>
    <from uri="direct:order"/>
    <marshal ref="jaxb"/> ← ❷ Transforms from model to XML
    <to uri="jms:queue:order"/>
  </route>
  <route>
    <from uri="jms:queue:order"/>
    <unmarshal ref="jaxb"/> ← ❸ Transforms from XML to model
    <to uri="direct:doSomething"/>
  </route>
</camelContext>

```

First you need to declare the JAXB data format ❶. 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 ❷, and the second route unmarshals to transform the XML back into the `PurchaseOrder` object ❸.

You can try this example by running the following Maven goal from the `chapter3/order` directory:

```
mvn test -Dtest=PurchaseOrderJaxbTest
```

NOTE To tell JAXB which classes are JAXB annotated, you need to drop a special `jaxb.index` file into each package in the classpath containing the POJO classes. It's a plain-text file in which each line lists the class name. In the preceding example, the file contains a single line with the text `PurchaseOrder`.

That's the basis of using XML object marshaling with `XStream` and JAXB. Both are implemented in Camel via data formats that are capable of transforming back and forth between various well-known formats.

3.4 Transforming with data formats

In Camel, data formats are pluggable transformers that can transform messages from one form to another, and vice versa. Each data format is represented in Camel as an interface in `org.apache.camel.spi.DataFormat` containing two methods:

- `marshal`—For marshaling a message into another form, such as marshaling Java objects to XML, CSV, JSON, HL7, or other well-known data models
- `unmarshal`—For performing the reverse operation, which turns data from well-known formats back into a message

You may already have realized that these two functions are opposites; one is capable of reversing what the other has done, as illustrated in figure 3.6.

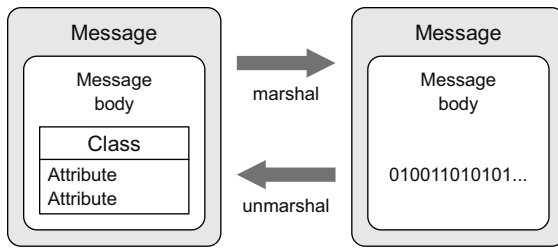


Figure 3.6 An object is marshaled to a binary representation; `unmarshal` can be used to get the object back.

We touched on data formats in section 3.3, where we covered XML transformations. This section covers data formats in more depth and using data types other than XML, such as CSV and JSON. We'll even look at how to create your own data formats. We'll start our journey by briefly looking at the data formats Camel provides out of the box.

3.4.1 Data formats provided with Camel

Camel provides data formats for a range of well-known data models, some of which are listed in table 3.3.

Table 3.3 Selection of data formats provided out of the box with Camel

Data format	Data model	Artifact	Description
Avro	Binary Avro format	camel-avro	Supports serializing and deserializing messages by using Apache Avro
Base64	Base64 string	camel-base64	Can encode and decode into a base64 string
Bindy	CSV, FIX, fixed length	camel-bindy	Binds various data models to model objects by using annotations
Crypto	Any	camel-crypto	Encrypts and decrypts data by using the Java Cryptography Extension
CSV	CSV	camel-csv	Transforms to and from CSV by using the Apache Commons CSV library
GSON	JSON	camel-gson	Transforms to and from JSON by using the Google GSON library
GZip	Any	camel-gzip	Compresses and decompresses files (compatible with the popular <code>gzip/gunzip</code> tools)
HL7	HL7	camel-hl7	Transforms to and from HL7, which is a well-known data format in the health-care industry
JAXB	XML	camel-jaxb	Uses the JAXB 2.x standard for XML binding to and from Java objects
Jackson	JSON	camel-jackson	Transforms to and from JSON by using the ultra-fast Jackson library
PGP	Any	camel-crypto	Encrypts and decrypts data by using PGP

Table 3.3 Selection of data formats provided out of the box with Camel (*continued*)

Data format	Data model	Artifact	Description
Protobuf	XML	camel-protobuf	Transforms to and from XML by using the Google Protocol Buffers library
SOAP	XML	camel-soap	Transforms to and from SOAP
Serialization	Object	camel-core	Uses Java Object Serialization to transform objects to and from a serialized stream
Syslog	RFC3164, RFC5424	camel-syslog	Transforms between RFC3164/RFC5424 messages and SyslogMessage model objects
XMLSecurity	XML	camel-xmlsecurity	Facilitates encryption and decryption of XML documents
XStream	XML	camel-xstream	Uses XStream for XML binding to and from Java objects
XStream	JSON	camel-xstream	Transforms to and from JSON by using the XStream library
Zip	Any	camel-core	Compresses and decompresses messages, and is most effective when dealing with large XML- or text-based payloads
Zip file	Zip file	camel-zipfile	Compresses and decompresses zip files

Camel provides more than 40 data formats out of the box. You can read more about these data formats at the Camel website (<http://camel.apache.org/data-format.html>). We've picked three to cover in the following section. They're among the most commonly used, and what you learn about those will also apply to the remainder of the data formats.

3.4.2 Using Camel's CSV data format

The camel-csv data format is capable of transforming to and from CSV format. It uses Apache Commons CSV to do the work.

Suppose you need to consume CSV files, split out each row, and send it to a JMS queue. Sounds hard to do, but it's possible with little effort in a Camel route:

```
from("file://rider/csvfiles")
    .unmarshal().csv()
    .split(body()).to("jms:queue:csv.record");
```

All you have to do is unmarshal the CSV files, which will read the file line by line and store all lines in the message body as a `java.util.List<List>` type. Then you use the splitter to split up the body, which will break the `java.util.List<List<String>>` into rows (each row represented as another `List<String>` containing the fields) and send each row to the JMS queue. You may not want to send each row as a `List` type to the JMS queue, so you can transform the row before sending, perhaps using a processor.

The same example in XML is a bit different, as shown here:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://rider/csvfiles"/>
    <unmarshal><csv/></unmarshal>
    <split>
      <simple>body</simple>
      <to uri="jms:queue:csv.record"/>
    </split>
  </route>
</camelContext>
```

The noticeable difference is in the way you tell `<split>` that it should split up the message body. To do this, you need to provide `<split>` with an `Expression`, which is what the splitter should iterate when it performs the splitting. To do so, you can use Camel's built-in expression language called `Simple` (see appendix A), which knows how to do that.

NOTE The Splitter EIP is fully covered in chapter 5.

This example is in the source code for the book, in the `chapter3/order` directory. You can try the examples by running the following Maven goals:

```
mvn test -Dtest=PurchaseOrderCsvTest
mvn test -Dtest=PurchaseOrderCsvSpringTest
```

At first, the data types that the CSV data format uses may seem confusing. They're listed in table 3.4.

Table 3.4 Data types that camel-csv uses when transforming to and from CSV format

Operation	From type	To type	Description
marshal	Map<String, Object>	OutputStream	Contains a single row in CSV format.
marshal	List<Map<String, Object>>	OutputStream	Contains multiple rows in CSV format; each row is separated by \n (newline).
unmarshal	InputStream	List<List<String>>	Contains a List of rows; each row is another List of fields.

One problem with `camel-csv` is that it uses generic data types, such as `Map` or `List`, to represent CSV records. Often you'll already have model objects to represent your data in memory. Let's look at using model objects with the `camel-bindy` component.

3.4.3 Using Camel's Bindy data format

Two of the existing CSV-related data formats (`camel-csv` and `camel-flatpack`) are older libraries that don't take advantage of the new features in Java 1.5, such as annotations

and generics. In light of this deficiency, Charles Moulliard stepped up and wrote the camel-bindy component to take advantage of these new possibilities. It's capable of binding CSV, FIX, and fixed-length formats to existing model objects by using annotations. This is similar to what JAXB does for XML.

Suppose you have a model object that represents a purchase order. By annotating the model object with camel-bindy annotations, you can easily transform messages between CSV and Java model objects, as shown in the following listing.

Listing 3.7 Model object annotated for CSV transformation

```
package camelinaction.bindy;

import java.math.BigDecimal;
import org.apache.camel.dataformat.bindy.annotation.CsvRecord;
import org.apache.camel.dataformat.bindy.annotation.DataField;

@CsvRecord(separator = ",", crlf = "UNIX") ← ❶ Maps to CSV record
public class PurchaseOrder {
    @DataField(pos = 1) ← ❷ Maps to column in CSV record
    private String name;
    @DataField(pos = 2, precision = 2) ← ❷ Maps to column in CSV record
    private BigDecimal price;
    @DataField(pos = 3) ← ❷ Maps to column in CSV record
    private int amount;
}
```

First you mark the class with the `@CsvRecord` annotation ❶ 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 ❷. Using the `pos` attribute, you can dictate the order in which they're output in CSV; `pos` starts with a value of 1. For numeric fields, you can additionally declare `precision`, which in this example is set to 2, indicating that the price should use two digits for cents. Bindy also has attributes for fine-grained layout of the fields, such as `pattern`, `trim`, and `length`. You can use `pattern` to indicate a data pattern, `trim` to trim the input, and `length` to restrict a text description to a certain number of characters.

Before you look at how to use Bindy in Camel routes, the data types Bindy expects to use are listed in table 3.5.

Table 3.5 Data types that Bindy uses when transforming to and from CSV format

Operation	From type	To type	Output description
marshal	List<Map<String, Object>>	OutputStream	Contains multiple rows in CSV format; each row is separated by \n (newline).
unmarshal	InputStream	List<Map<String, Object>>	Contains a List of rows; each row contains 1 ... n data models contained in a Map.

The important thing to notice in table 3.5 is that Bindy uses `Map<String, Object>` to represent a CSV row. At first, this may seem odd. Why doesn't it use a single model object for that? The answer is that you can have multiple model objects with the CSV record being scattered across those objects. For example, you could have fields 1 to 3 in one model object, fields 4 to 9 in another, and fields 10 to 12 in a third.

The map entry `<String, Object>` is distilled as follows:

- 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 confusing, don't worry. The following listing should make it clearer.

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

```
public void testBindy() throws Exception {
    CamelContext context = new DefaultCamelContext();
    context.addRoutes(createRoute());
    context.start();
    MockEndpoint mock = context.getEndpoint("mock:result",
                                           MockEndpoint.class);
    mock.expectedBodiesReceived("Camel in Action,69.99,1\n");
    PurchaseOrder order = new PurchaseOrder();
    order.setAmount(1);
    order.setPrice(new BigDecimal("69.99"));
    order.setName("Camel in Action");
    ProducerTemplate template = context.createProducerTemplate();
    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)
                .to("mock:result");
        }
    };
}
```

① Creates model object as usual

② Starts test

③ Transforms model object to CSV

In listing 3.8, you first create and populate the order model by using regular Java setters ①. Then you send the order model to the route by sending it to the `direct:toCsv` endpoint ② used in the route. The route will then marshal the order model to CSV by using Bindy ③. Notice that Bindy is configured to use CSV mode via `BindyType.Csv`. To let Bindy know how to map the order model object, you need to provide a class annotated with Bindy annotations, as in listing 3.7.

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

You can try this example from the `chapter3/order` directory by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderBindyTest
```

The source code for the book also contains a *reverse* example of how to use Bindy to transform a CSV record into a Java object. You can try it by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderUnmarshalBindyTest
```

CSV is only one of the well-known data formats that Bindy supports. Bindy is equally capable of working with fixed-length and FIX data formats, both of which follow the same principles as CSV.

It's now time to leave CSV and look at a more modern format: JSON.

3.4.4 Using Camel's JSON data format

JavaScript Object Notation (JSON) is a data-interchange format, and Camel provides six components that support the JSON data format: `camel-xstream`, `camel-gson`, `camel-jackson`, `camel-boon`, `camel-fastjson`, `camel-johnzon`. This section focuses on `camel-jackson` because Jackson is a popular JSON library.

Back at Rider Auto Parts, you now have to implement a new service that returns order summaries rendered in JSON format. Doing this with Camel is fairly easy, because Camel has all the ingredients needed to brew this service. The following listing shows how to ramp up a prototype.

Listing 3.9 An HTTP service that returns order summaries rendered in JSON format

```
<bean id="orderService" class="camelinaction.OrderServiceBean"/>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <dataFormats>
    <json id="json" library="Jackson"/>
  </dataFormats>
  <route>
    <from uri="jetty://http://0.0.0.0:8080/order"/>
    <bean ref="orderService" method="lookup"/>
    <marshal ref="json"/>
  </route>
</camelContext>
```

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 ❷, and the `lookup` method is invoked on that bean. The result of this bean invocation is then marshaled to JSON format and returned to the HTTP client.

The order service bean could have a method signature such as this:

```
public PurchaseOrder lookup(@Header(name = "id") String id)
```

This signature allows you to implement the lookup logic as you wish. You'll learn more about the `@Header` annotation in chapter 4, when we cover how bean parameter binding works in Camel.

Notice that the service bean can return a POJO that the JSON library is capable of marshaling. For example, suppose you used the `PurchaseOrder` from listing 3.7 and had JSON output as follows:

```
{"name": "Camel in Action", "amount": 1.0, "price": 69.99}
```

The HTTP service itself can be invoked by an HTTP Get request with the `id` of the order as a parameter: `http://0.0.0.0:8080/order/service?id=123`.

Notice how easy it is with Camel to bind the HTTP `id` parameter as the `String id` parameter with the help of the `@Header` annotation.

You can try this example yourself from the `chapter3/order` directory by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderJSONTest
```

So far, we've used data formats with their default settings. But what if you need to configure the data format, for example, to use another splitter character with the CSV data format? That's the topic of the next section.

3.4.5 **Configuring Camel data formats**

In section 3.4.2, you used the CSV data format, but this data format offers many additional settings. The following listing shows how to configure the CSV data format.

Listing 3.10 Configuring the CSV data format

```
public void configure() {
    CsvDataFormat myCsv = new CsvDataFormat()
        .setDelimiter(';')
        .setHeader(new String[] {
            "id", "customerId", "date", "item", "amount", "description"});
    from("direct:toCsv")
        .marshal(myCsv)
        .to("file://acme/outbox/csv");
}
```

❶ Creates and configures a custom CSV data format

❷ Uses CSV data format

Configuring data formats in Camel is typically done directly on `DataFormat` itself; sometimes you may also need to use the API that the third-party library under the hood provides. In listing 3.10, the CSV data format nicely wraps the third-party API so you can just configure the `DataFormat` directly ❶. Here you set the semicolon as a delimiter and specify the order of the fields ❶. The use of the data format stays the

same, so all you need to do is refer to it from the `marshal` **2** or `unmarshal` methods. This same principle applies to all data formats in Camel.

TIP You can learn how to create your own data format in chapter 8, section 8.5.

You've learned all about data formats, and now it's time to say goodbye to data formats and take a look at using templating with Camel for data transformation. Templating is extremely useful when you need to generate automatic reply emails.

3.5 Transforming with templates

Camel provides slick integration with two template languages:

- *Apache Velocity*—Probably the best-known templating language (<http://camel.apache.org/velocity.html>)
- *Apache FreeMarker*—Another great templating language from Apache (<http://camel.apache.org/freemarker.html>)

These two templating languages are fairly similar to use, so we discuss only Velocity here.

3.5.1 Using Apache Velocity

Rider Auto Parts has implemented a new order system that must send an email reply when a customer has submitted an order. Your job is to implement this feature.

The reply email could look like this:

```
Dear customer
Thank you for ordering X piece(s) of XXX at a cost of XXX.
This is an automated email, please do not reply.
```

Three pieces of information in the email must be replaced at runtime with real values. You need to adjust the email to use the Velocity template language, and then place it into the source repository as `src/test/resources/email.vm`:

```
Dear customer
Thank you for ordering ${body.amount} piece(s) of ${body.name} at a cost of
    ${body.price}.
This is an automated email, please do not reply.
```

Notice that you insert `${ }` placeholders in the template, which instructs Velocity to evaluate and replace them at runtime. Camel prepopulates the Velocity context with numerous entities that are then available to Velocity. Those entities are listed in table 3.6.

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

Table 3.6 Entities that are prepopulated in the Velocity context and available at runtime

Entity	Type	Description
<code>camelContext</code>	<code>org.apache.camel.CamelContext</code>	The <code>CamelContext</code> .
<code>exchange</code>	<code>org.apache.camel.Exchange</code>	The current exchange.

Table 3.6 Entities that are prepopulated in the Velocity context and available at runtime (*continued*)

Entity	Type	Description
in	org.apache.camel.Message	The input message. This can clash with a reserved word in some languages; use request instead.
request	org.apache.camel.Message	The input message.
body	java.lang.Object	The input message body.
headers	java.util.Map	The input message headers.
response	org.apache.camel.Message	The output message.
out	org.apache.camel.Message	The output message. This can clash with a reserved word in some languages; use response instead.

Using Velocity in a Camel route is as simple as this:

```
from("direct:sendMail")
    .setHeader("Subject", constant("Thanks for ordering"))
    .setHeader("From", constant("donotreply@riders.com"))
    .to("velocity://rider/mail.vm")
    .to("smtp://mail.riders.com?user=camel&password=secret");
```

All you have to do is route the message to the Velocity endpoint that's configured with the template you want to use, which is the rider/mail.vm file that's loaded from the classpath by default. All the template components in Camel use the same resource loader, which allows you to load templates from the classpath, file paths, and other such locations. You can use the same prefixes listed in table 3.2.

You can try this example by going to the chapter3/order directory in the book's source code and running the following Maven goal:

```
mvn test -Dtest=PurchaseOrderVelocityTest
```

We'll now leave data transformation and look at type conversion. Camel has a powerful type-converter mechanism that removes all need for boilerplate type-converter code.

3.6 **Understanding Camel type converters**

Camel provides a built-in type-converter system that automatically converts between well-known types. This system allows Camel components to easily work together without having type mismatches. And from the Camel user's perspective, type conversions are built into the API in many places without being invasive. For example, you used it in listing 3.1:

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

The `getBody` method is passed the type you want to have returned. Under the covers, the type-converter system converts the returned type to a `String` if needed.

In this section, you'll take a look at the insides of the type-converter system. We'll explain how Camel scans the classpath on startup to register type converters dynamically. We'll also show how to use it from a Camel route, and how to build your own type converters.

3.6.1 How the Camel type-converter mechanism works

To understand the type-converter system, you first need to know what a type converter in Camel is. Figure 3.7 illustrates the relationship between `TypeConverterRegistry` and the `TypeConverters` it holds.

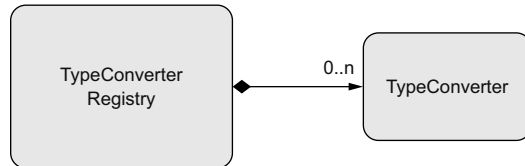


Figure 3.7 The `TypeConverterRegistry` contains many `TypeConverters`

`TypeConverterRegistry` is where all the type converters are registered when Camel is started. At runtime, Camel uses the `TypeConverterRegistry`'s `lookup` method to look up a suitable `TypeConverter`:

```
TypeConverter lookup(Class<?> toType, Class<?> fromType);
```

By using `TypeConverter`, Camel can then convert one type to another by using `TypeConverter`'s `convertTo` method, which is defined as follows:

```
<T> T convertTo(Class<T> type, Object value);
```

NOTE Camel implements about 350 or more type converters out of the box, which are capable of converting to and from the most commonly used types.

LOADING TYPE CONVERTERS INTO THE REGISTRY

On startup, Camel loads all the type converters into the `TypeConverterRegistry` by using a classpath-scanning solution. This allows Camel to pick up type converters not only from `camel-core`, but also from any of the other Camel components, including your Camel applications. You'll see this in section 3.6.3 when you build your own type converter.

Camel uses `org.apache.camel.impl.converter.AnnotationTypeConverterLoader` to scan and load the type converters. To avoid scanning zillions of classes, it reads a service discovery file in the `META-INF` folder: `META-INF/services/org/apache/camel/TypeConverter`. This is a plain-text file that has a list of fully qualified class names and packages that contain Camel type converters. The special file is needed to avoid scanning every possible JAR and all their packages, which would be time-consuming. This special file tells Camel whether the JAR file contains type converters. For example, the file in `camel-cxf` contains the following entries:

```
org.apache.camel.component.cxf.converter.CxfConverter
```

```
org.apache.camel.component.cxf.converter.CxfPayloadConverter
```

`AnnotationTypeConverterLoader` loads those classes that have been annotated with `@Converter`, and then searches within them for public methods that are annotated with `@Converter`. Each of those methods is considered a type converter. Yes, the class `@Converter` annotation is a bit of overkill when we've already defined the class name in the `TypeConverter` text file. We need this because we can also specify package names, which could include many classes. For example, a package name of `org.apache.camel.component.cxf.converter` also could have been provided in the `TypeConverter` text file and would have included `CxfConverter` and `CxfPayloadConverter`. Using the fully qualified class name is preferred, though, because Camel loads them more quickly.

This process is best illustrated with an example. The following code is a snippet from the `IOConverter` class from the `camel-core` JAR:

```
@Converter
public final class IOConverter {
    @Converter
    public static InputStream toInputStream(URL url) throws IOException {
        return IOHelper.buffered(url.openStream());
    }
}
```

Camel will go over each method annotated with `@Converter` and look at the method signature. The first parameter is the *from* type, and the return type is the *to* type. In this example, you have a `TypeConverter` that can convert from a `URL` to an `InputStream`. By doing this, Camel loads all the built-in type converters, including those from the Camel components in use.

TIP Type converters can also be loaded into the registry manually. This is often useful if you need to quickly add a type converter into your application or want full control over when it will be loaded. You can find more information in the online documentation (<http://camel.apache.org/type-converter.html>).

Now that you know how the Camel type converters are loaded, let's look at using them.

3.6.2 *Using Camel type converters*

As we mentioned, the Camel type converters are used throughout Camel, often automatically. But you might want to use them to force a specific type to be used in a route, such as before sending data back to a caller or a JMS destination. Let's look at how to do that.

Suppose you need to route files to a JMS queue by using `javax.jms.TextMessage`. To do so, you can convert each file to a `String`, which forces the JMS component to use `TextMessage`. This is easy to do in Camel—you use the `convertBodyTo` method, as shown here:

```
from("file://riders/inbox")
    .convertBodyTo(String.class)
```

```
.to("jms:queue:inbox");
```

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

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

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

Another reason for using `convertBodyTo` is to read files by using a fixed encoding such as UTF-8. This is done by passing in the encoding as the second parameter:

```
from("file://riders/inbox")
  .convertBodyTo(String.class, "UTF-8")
  .to("jms:queue:inbox");
```

TIP If you have trouble with a route because of the payload or its type, try using `.convertBodyTo(String.class)` at the start of the route to convert to a `String` type, which is a well-supported type. If the payload can't be converted to the desired type, a `NoTypeConversionAvailableException` exception is thrown.

That's all there is to using type converters in Camel routes. Before we wrap up this chapter, though, let's take a look at how to write your own type converter.

3.6.3 Writing your own type converter

Writing your own type converter is easy in Camel. You already saw what a type converter looks like in section 3.6.1, when you looked at how type converters work.

Suppose you want to write a custom type converter that can convert a `byte []` into a `PurchaseOrder` model object (an object you used in listing 3.7). As you saw earlier, you need to create an `@Converter` class containing the type-converter method, as shown in the following listing.

Listing 3.11 A custom type converter to convert from `byte []` to `PurchaseOrder` type

```
@Converter
public final class PurchaseOrderConverter
  @Converter
  public static PurchaseOrder toPurchaseOrder(byte [] data,
                                             Exchange exchange) {
    TypeConverter converter = exchange.getContext()
                                     .getTypeConverter();
    String s = converter.convertTo(String.class, data);
    if (s == null || s.length() < 30) {
      throw new IllegalArgumentException("data is invalid");
    }
  }
```

1 Grabs TypeConverter to reuse

```

    s = s.replaceAll("##START##", "");
    s = s.replaceAll("##END##", "");
    String name = s.substring(0, 9).trim();

    String s2 = s.substring(10, 19).trim();
    BigDecimal price = new BigDecimal(s2);

    price.setScale(2);
    String s3 = s.substring(20).trim();
    Integer amount = converter
        .convertTo(Integer.class, s3);
    return new PurchaseOrder(name, price, amount);
}
}

```

② Converts from String to PurchaseOrder

The Exchange gives you access to the CamelContext and thus to the parent TypeConverter ①, which you use in this method to convert between strings and numbers. The rest of the code is the logic for parsing the custom protocol and returning the PurchaseOrder ②. Notice that you can use converter to easily convert between well-known types.

All you need to do now is add the service discovery file, named TypeConverter, in the META-INF directory. As explained previously, this file contains the fully qualified name of the @Converter class.

If you cat the TypeConverter file, you'll see this:

```

$ cat src/main/resources/META-INF/services/org/apache/camel/TypeConverter
camelinaction.PurchaseOrderConverter

```

This example can be found in the chapter3/converter directory of the book's source code, which you can try by using the following Maven goal:

```

mvn test -Dtest=PurchaseOrderConverterTest

```

RETURNING NULL VALUES

By default, a null return value from a type converter isn't valid. Camel considers null as a "miss" and adds the pair of types you're trying to convert to a blacklist so they won't be tried again. For example, if our previous example returned null, the conversion from byte[] to PurchaseOrder would be blacklisted. If null is a valid return value for your conversion, you can force Camel to accept it by using the allowNull option on the @Converter annotation. For example, if the example in listing 3.11 required a null return value, you could do something like this:

```

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

```

ADDING TYPE CONVERTERS TO CAMEL-CORE

If you're on track to becoming a star Camel rider and want to write a shiny new type converter for the camel-core module, you may notice that type-converter loading is handled differently there. The META-INF/services/org/apache/camel/TypeConverter file specifies the org.apache.camel.core package, which doesn't exist. It's just a dummy

package name. The type converters for camel-core are specified directly in `org.apache.camel.impl.converter.CorePackageScanClassResolver`, and you can add them there. And that completes this chapter on transforming data with Camel.

3.7 Summary and best practices

Data transformation is the cornerstone of any integration kit; it bridges the gap between various data types and formats. It's also essential in today's industry, because more and more disparate systems need to be integrated to support the ever-changing businesses and world we live in.

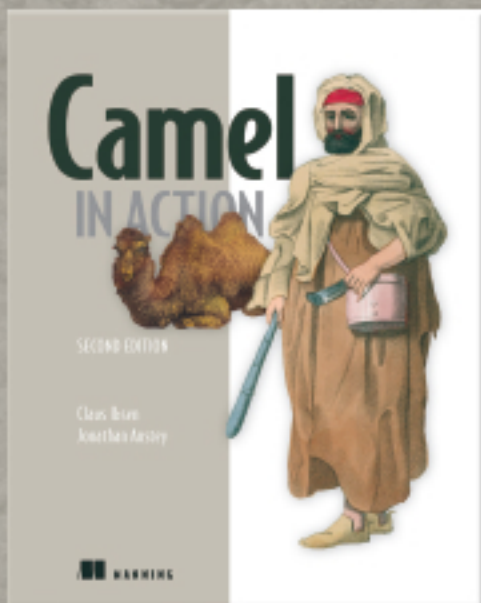
This chapter covered many of the possibilities Camel offers for data transformation. You learned how to format messages by using EIPs and beans. You also learned that Camel provides special support for transforming XML documents by using XSLT components and XML-capable data formats. Camel provides data formats for well-known data models, which you learned to use, and it even allows you to build your own data formats. We also took a look into the templating world, which can be used to format data in specialized cases, such as generating email bodies. Finally, we looked at how the Camel type-converter mechanism works and learned that it's used internally to help all the Camel components work together. You learned how to use it in routes and how to write your own converters.

Here are a few key tips you should take away from this chapter:

- *Data transformation is often required*—Integrating IT systems often requires you to use different data formats when exchanging data. Camel can act as the mediator and has strong support for transforming data in any way possible. Use the various features in Camel to aid with your transformation needs.
- *Java is powerful*—Using Java code isn't a worse solution than using a fancy mapping tool. Don't underestimate the power of the Java language. Even if it takes 50 lines of grunt boilerplate code to get the job done, you have a solution that can easily be maintained by fellow engineers.
- *Prefer to use beans over processors*—If you're using Java code for data transformation, you can use beans or processors. Processors are more dependent on the Camel API, whereas beans allow loose coupling. Chapter 4 covers how to use beans.

This chapter, along with chapter 2, covered two crucial features of integration kits: routing and transformation. The next chapter dives into the world of Java beans, and you'll see how Camel can easily adapt to and use your existing beans. This allows a higher degree of reuse and loose coupling, so you can keep your business and integration logic clean and apart from Camel and other middleware APIs.

Compliments of  RED HAT
DEVELOPER
PROGRAM



Special discount!

Buy **Camel in Action, Second Edition**
for **50%** off with discount code
camelred at <http://bit.ly/IPwr1VI>

Selections from Camel in Action Second Edition
Claus Ibsen and Jonathan Anstey

 manning