Camel IN ACTION

SECOND EDITION

Claus Ibsen
Jonathan Anstey

MANNING

MEAP Edition
Manning Early Access Program

Camel in Action

Second Edition
Version 13

welcome

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

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

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

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

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

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

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

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

Regards

—Claus Ibsen and Jonathan Anstey

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

brief contents

Part 1

First steps

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

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

1 Meeting Camel

This chapter covers

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

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

- You can't finalize a jigsaw puzzle until you have a complete set of pieces that plug into each other simply, seamlessly, and robustly. That holds true for system integration projects as well. But whereas jigsaw puzzle pieces are made to plug into each other, the systems we integrate rarely are. Integration frameworks aim to fill this gap. As an integrator, you're less concerned about how the system you integrate works and more focused on how to interoperate with it from the outside. A good integration framework provides simple, manageable abstractions for the complex systems you're integrating and the "glue" for plugging them together seamlessly.
- Apache Camel is such an integration framework. In this book, we'll help you understand what Camel is, how to use it, and why we think it's one of the best integration frameworks out there.

This chapter will start off by introducing Camel and highlighting some of its core features. We'll then take a look at the Camel distribution and explain how you can run the Camel examples in the book. We'll round off the chapter by bringing core Camel concepts to the table so you can understand Camel's architecture.

Are you ready? Let's meet Camel.

1.1 Introducing Camel

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

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

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

1.1.1 What is Camel?

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

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

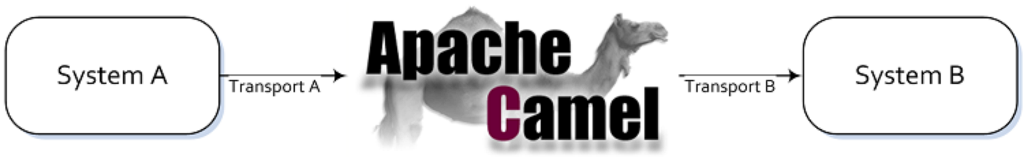


Figure 1.1 Camel is the glue between disparate systems.

Camel offers higher-level abstractions that allow you to interact with various systems using the same API regardless of the protocol or data type the systems are using. Components in Camel provide specific implementations of the API that target different protocols and data types. Out of the box, Camel comes with support for about 200 protocols and data types. Its extensible and modular architecture allows you to implement and seamlessly plug in support for your own protocols, proprietary or not. These architectural choices eliminate the need for unnecessary conversions and make Camel not only faster but also very lean. As a result, it's suitable for embedding into other projects that require Camel's rich processing capabilities. Other open source projects, such as Apache ServiceMix, Karaf and ActiveMQ, already use Camel as a way to carry out enterprise integration.

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

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

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

1.1.2 Why use Camel?

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

• **Routing and mediation engine**	• **Enterprise integration patterns (EIPs)**
• **Domain-specific language (DSL)**	• **Extensive component library**
• **Payload-agnostic router**	• **Modular and pluggable architecture**
• **POJO model**	• **Easy configuration**
• **Automatic type converters**	• **Lightweight core**
• **Test kit**	• **Vibrant community**

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

ROUTING AND MEDIATION ENGINE

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

EXTENSIVE COMPONENT LIBRARY

Camel provides an extensive library of about 200 components. These components enable Camel to connect over transports, use APIs, and understand data formats. Try to spot a few technologies that you've used in the past or want to use in the future in Figure 1.2.

google-calendar eventadmin
guava-eventbus imaps facebook
geocoder bean-validator aws-sqs gmail
jgroups https aws-ddb atom imap
aws-sns spring-integration atmos
elasticsearch grape
docker spring-redis solrCloud avro jbpm
jdbc aws-swf aws-cw http sjms-batch xmlsecurity spring-ldap binding ftp
cometds jcr schematron validator ghttp
direct scp ssh mustache spark-rest
controlbus stomp language kestrel metrics xquery coap aws-ec2
jira box seda netty-http salesforce stax smpps
class sip mongodb optaplanner mvel zookeeper cxf aws-ses
gauth bean cache snmp linkedin paxlogging rabbitmq lucene ibatis github
cql rss log smpp cometd
jetty cmis msv netty quartz2 pop3 pgevent openshift velocity browse
undertow irc
http4s ejb solr mina2 restlet pdf quartz ref kafka sips jmx git
apns sftp mqtt paho sql rest lpr test xmlrpc dataset gtask
jing direct-vm jms stream routebox pop3s netty4 rmi hipchat
jolt netty4-http ldap krati timer splunk ahc-wss
ahc spring-ws mybatis rest-apiquickfix sjms cxfrs hdfs
fop chunk mock
amqp slack twitter jpa sap-netweaver smtps
ftps http4 crypto vertx stub mina xslt servlet
jt400 dropbox infinispan
yammer nagios scheduler xmpp
ahc-ws spring-event
couchdb stringtemplate weather
jclouds aws-s3dozer smtp websocket solrs dns elsql google-mail
disruptor-vm spring-batch context aws-sdb glogin
exec dataformat disruptor hbase gora
beanstalk file
atmosphere-websocket
ganglia hazelcast hdfs2freemarker
javaspace flatpack
google-drive

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

ENTERPRISE INTEGRATION PATTERNS (EIPS)

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

The enterprise integration patterns, or EIPs, are helpful not only because they provide a proven solution for a given problem, but also because they help define and communicate the problem itself. Patterns have known semantics, which makes communicating problems much easier. The difference between using a pattern language and describing the problem at hand is similar to using spoken language rather than sign language. If you’ve ever visited a foreign country, you’ve probably experienced the difference.

Camel is heavily based on EIPs. Although EIPs describe integration problems and solutions and also provide a common vocabulary, the vocabulary isn’t formalized. Camel tries to close this gap by providing a language to describe the integration solutions. There’s almost a one-to-one relationship between the patterns described in *Enterprise Integration Patterns* and the Camel DSL.

DOMAIN-SPECIFIC LANGUAGE (DSL)

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

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

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

- XML DSL

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

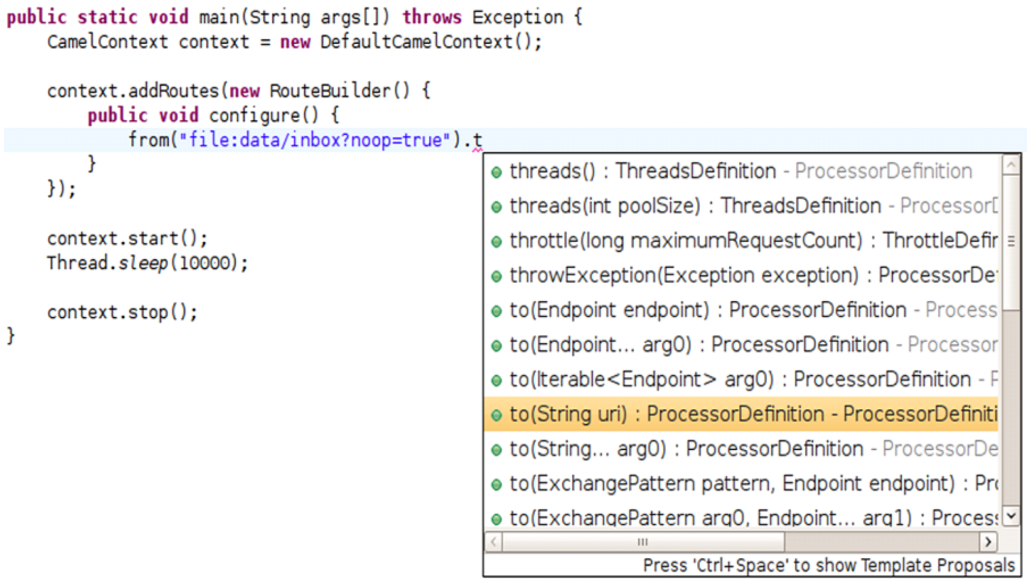
- Scala DSL

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

- Groovy DSL

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

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



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

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

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

    context.stop();
}
```

```
threads() : ThreadsDefinition - ProcessorDefinition
threads(int poolSize) : ThreadsDefinition - ProcessorI
throttle(long maximumRequestCount) : ThrottleDefir
throwException(Exception exception) : ProcessorDe
to(Endpoint endpoint) : ProcessorDefinition - Process
to(Endpoint... arg0) : ProcessorDefinition - Processor
to(Iterable<Endpoint> arg0) : ProcessorDefinition - F
to(String uri) : ProcessorDefinition - ProcessorDefiniti
to(String... arg0) : ProcessorDefinition - ProcessorDe
to(ExchangePattern pattern, Endpoint endpoint) : Pr
to(ExchangePattern arg0, Endpoint... arg1) : Proces
Press 'Ctrl+Space' to show Template Proposals
```

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

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

PAYLOAD-AGNOSTIC ROUTER

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

MODULAR AND PLUGGABLE ARCHITECTURE

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

POJO MODEL

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

EASY CONFIGURATION

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

For example, you could configure a file consumer to scan recursively in a subfolder and include only a .txt file, as follows:

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

AUTOMATIC TYPE CONVERTERS

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

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

LIGHTWEIGHT CORE

Camel's core can be considered pretty lightweight, with the total library coming in at about 3.3 MB and only having 1.4 MB of dependencies. This makes Camel easy to embed or deploy anywhere you like, such as in a standalone application, microservice, web application, Spring application, Java EE application, OSGi, Spring Boot, WildFly, and in cloud platforms such as Kubernetes and Cloud Foundry. Camel was designed not to be a server or ESB but instead to be embedded in whatever platform you choose. You just need Java.

TEST KIT

Camel provides a Test Kit that makes it easier for you to test your own Camel applications. The same Test Kit is used extensively to test Camel itself, and it includes more than 16,000 unit tests. The Test Kit contains test-specific components that, for example, can help you mock real endpoints. It also contains setup expectations that Camel can use to determine whether an application satisfied the requirements or failed. Chapter 9 covers testing with Camel.

VIBRANT COMMUNITY

Camel has an active community. It's a long-lived one too. It has been active (and growing) for over 9 years at the time of writing. Having a strong community is essential if you intend to use any open source project in your application. Inactive projects have little community support, so if you run into issues, you’re on your own. With Camel, if you’re having any trouble, users and developers alike will come to your aid promptly. For more information on Camel’s community, see appendix D.

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

1.2 Getting started

In this section, we’ll show you how to get your hands on a Camel distribution, explain what’s inside, and then run an example using Apache Maven. After this, you’ll know how to run any of the examples from the book’s source code.

Let’s first get the Camel distribution.

1.2.1 Getting Camel

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

- For the purposes of this book, we’ll be using Camel 2.17.0. To get this version, click on the Camel 2.17.0 Release link and near the bottom of the page you’ll find two binary distributions: the zip distribution is for Windows users, and the tar.gz distribution is for Mac OS X/Linux users. When you’ve downloaded one of the distributions, extract it to a location on your hard drive.
- Open up a command prompt, and go to the location where you extracted the Camel distribution. Issuing a directory listing here will give you something like this:

```
[janstey@bender apache-camel-2.17.0]$ ls
doc examples lib LICENSE.txt NOTICE.txt README.txt
```

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

- doc—Contains the Camel manual in HTML format. This manual is a download of a large portion of the Apache Camel wiki at the time of release. As such, it’s a decent reference for those not able to access the Camel website (or if you misplaced your copy of Camel in Action).
- examples—Includes 64 Camel examples. You’ll see an example shortly.
- lib—Contains all Camel libraries and third-party dependencies needed for the core of Camel to run. You’ll see later in the chapter how Maven can be used to easily grab dependencies for the components outside the core.
- LICENSE.txt—Contains the license of the Camel distribution. Because this is an Apache project, the license is the Apache License, version 2.0.
- NOTICE.txt—Contains copyright information about the third-party dependencies

included in the Camel distribution.

- README.txt—Contains a short intro to what Camel is and a list of helpful links to get new users up and running fast.

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

1.2.2 Your first Camel ride

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

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

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

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

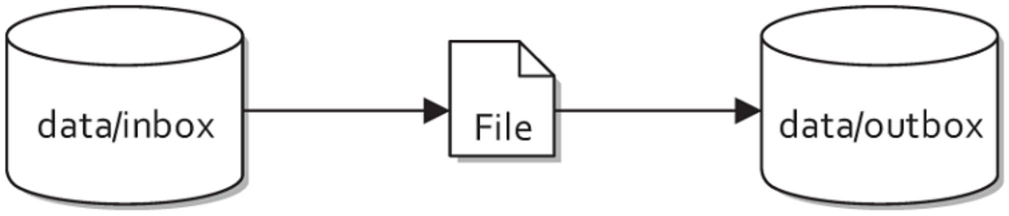


Figure 1.4 Files are routed from the data/inbox directory to the data/outbox directory.

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

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

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

public class FileCopier {
    public static void main(String args[]) throws Exception {
        File inboxDirectory = new File("data/inbox");
        File outboxDirectory = new File("data/outbox");
        outboxDirectory.mkdir();
        File[] files = inboxDirectory.listFiles();
        for (File source : files) {
            if (source.isFile()) {
               File dest = new File(
                    outboxDirectory.getPath()
                    + File.separator
                    + source.getName());
               copyFile(source, dest);
            }
        }
    }
    private static void copyFile(File source, File dest)
        throws IOException {
        OutputStream out = new FileOutputStream(dest);
        byte[] buffer = new byte[(int) source.length()];
        FileInputStream in = new FileInputStream(source);
        in.read(buffer);
        try {
            out.write(buffer);
        } finally {
            out.close();
            in.close();
        }
    }
}
```

The `FileCopier` example in listing 1.1 is a pretty simple use case, but it still results in 34 lines of code. You have to use low-level file APIs and ensure that resources get closed properly, a task that can easily go wrong. Also, if you wanted to poll the data/inbox directory for new files, you'd need to set up a timer and also keep track of which files you've already copied. This simple example is getting more complex.

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

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

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

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

(1) Routes files from inbox to outbox

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

Routes in Camel are defined in such a way that they flow when read. This route can be read like this: consume messages `from file` location `data/inbox` with the `noop` option set, and send `to file` location `data/outbox`. The `noop` option tells Camel to leave the source file as is. If you didn't use this option, the file would be moved. Most people who have never seen Camel before will be able to understand what this route does. You may also want to note that, excluding the boilerplate code, you created a file-polling route in just one line of Java code (1) .

To run this example, you'll need to download and install Apache Maven from the Maven site at http://maven.apache.org/download.html. If you prefer to stay within your IDE, you can find setup instructions in Appendix A. Once you have Maven up and working, open a terminal and browse to the chapter1/file-copy directory of the book's source. If you take a directory listing here, you'll see several things:

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

NOTE We used Maven 3.3.3 during the development of the book. Different versions of Maven may not work or appear exactly as we've shown.

The POM is shown here.

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

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.camelinaction</groupId>         (1)
    <artifactId>chapter1</artifactId>            (1)
    <version>2.0.0</version>                     (1)
  </parent>
  <artifactId>chapter1-file-copy</artifactId>
  <name>Camel in Action 2 :: Chapter 1 :: File Copy Example</name>

  <dependencies>
    <dependency>
      <groupId>org.apache.camel</groupId>       ❷
      <artifactId>camel-core</artifactId>       ❷
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>              ❸
      <artifactId>slf4j-log4j12</artifactId>    ❸
    </dependency>
  </dependencies>
</project>
```

❶ **Parent POM**
❷ **Camel's core library**
❸ **Logging support**

Maven itself is a complex topic, and we won't go into great detail here. We'll give you enough information to be productive with the examples in this book. For an in-depth look at Maven, we recommend reading *Maven by Example* and *Maven: The Complete Reference*, both of which are freely available from http://books.sonatype.com. We'll also discuss using Maven to develop Camel applications in chapter 8, so there's a good deal of information there too.

The Maven POM in listing 1.3 is probably one of the shortest POMs you'll ever see—almost everything uses the defaults provided by Maven. Besides those defaults, there are also some settings configured in the parent POM ❶ . Probably the most important section to point out here is the dependency on the Camel library ❷ . 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 ❶ , and will resolve to 2.17.0. The type of dependency was not specified, so the JAR file type will be assumed. The search path will be org/apache/camel/camel-core/2.17.0/camel-core-2.17.0.jar.
2. Because listing 1.3 defined no special places for Maven to look for the Camel dependencies, it will look in Maven's central repository, located at http://repo1.maven.org/maven2.
3. Combining the search path and the repository URL, Maven will try to download http://repo1.maven.org/maven2/org/apache/camel/camel-core/2.17.0/camel-core-2.17.0.jar.
4. This JAR will be saved to Maven's local download cache, which is typically located in the home directory under .m2/repository. This would be ~/.m2/repository on Linux/Mac OS X and C:\Users\<Username>\.m2\repository on recent versions of Windows.
5. When the application code in listing 1.2 is started, the Camel JAR will be added to the classpath.

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

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

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

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

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

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

1.3 Camel's message model

In Camel, there are two abstractions for modeling messages, both of which we'll cover in this section:

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

We'll start by looking at Message to understand how data is modeled and carried in Camel. Then we'll look at how a "conversation" is modeled in Camel by the Exchange.

1.3.1 Message

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

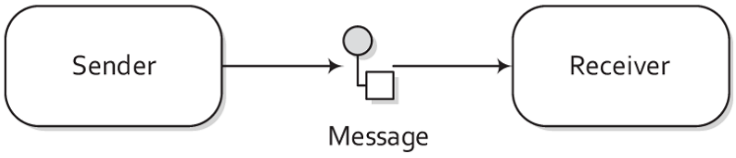


Figure 1.5 Messages are entities used to send data from one system to another.

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

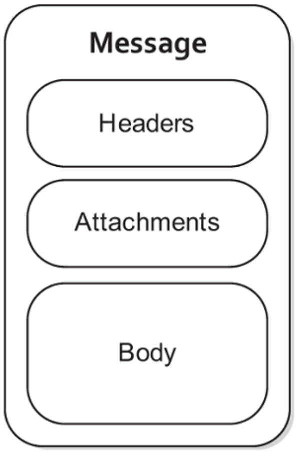


Figure 1.6 A message can contain headers, attachments, and a body.

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

HEADERS AND ATTACHMENTS

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

BODY

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

FAULT FLAG

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

During routing, messages are contained in an exchange.

1.3.2 Exchange

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

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

Figure 1.7 illustrates the contents of an exchange in Camel.

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

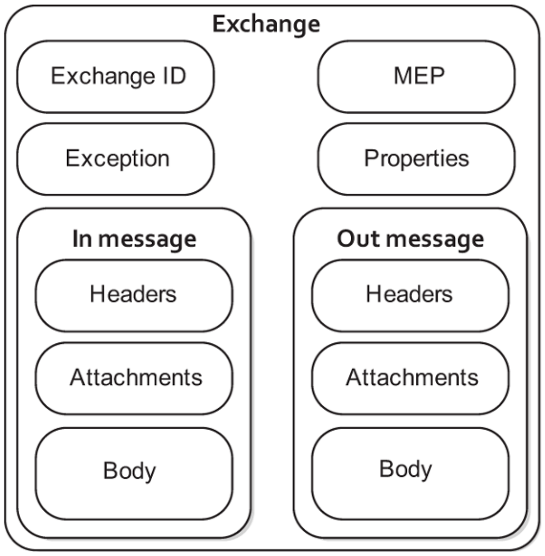


Figure 1.7 A Camel exchange has an ID, MEP, exception, and properties. It also has an *in* message to store the incoming message and an *out* message to store the result.

- *Exchange ID*—A unique ID that identifies the exchange. Camel will generate a default unique ID, if you don't explicitly set one.
- *MEP*—A pattern that denotes whether you're using the `InOnly` or `InOut` messaging style. When the pattern is `InOnly`, the exchange contains an in message. For `InOut`, an out message also exists that contains the reply message for the caller.
- *Exception*—If an error occurs at any time during routing, an `Exception` will be set in the exception field.
- *Properties*—Similar to message headers, but they last for the duration of the entire exchange. Properties are used to contain global-level information, whereas message headers are specific to a particular message. Camel itself will add various properties to the exchange during routing. You, as a developer, can store and retrieve properties at any point during the lifetime of an exchange.
- *In message*—This is the input message, which is mandatory. The in message contains the request message.
- *Out message*—This is an optional message that only exists if the MEP is `InOut`. The out message contains the reply message.

The Exchange is the same for the entire life cycle of routing, but the messages can change. For instance, if messages are transformed from one format to another.

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

1.4 Camel's architecture

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

1.4.1 Architecture from 10,000 feet

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

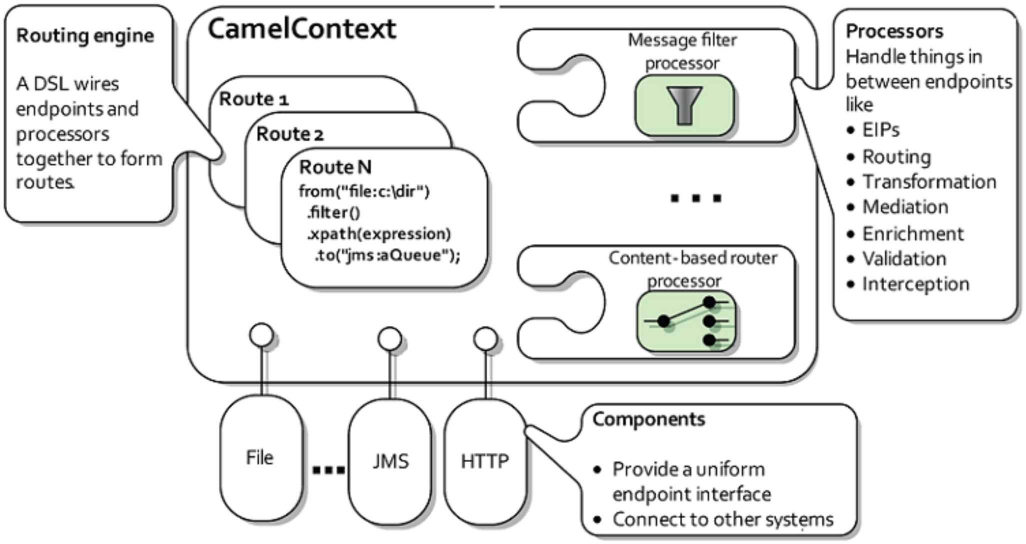


Figure 1.8 At a high level, Camel is composed of processors, components, and routes. All of these are contained within the `CamelContext`.

The routing engine uses routes as specifications for where messages are routed. Routes are defined using one of Camel's domain-specific languages (DSLs). Processors are used to transform and manipulate messages during routing and also to implement all the EIP patterns, which have corresponding keywords in the DSL languages. Components are the extension points in Camel for adding connectivity to other systems. To expose these systems to the rest of Camel, components provide an endpoint interface.

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

1.4.2 Camel concepts

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

CAMELCONTEXT

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

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

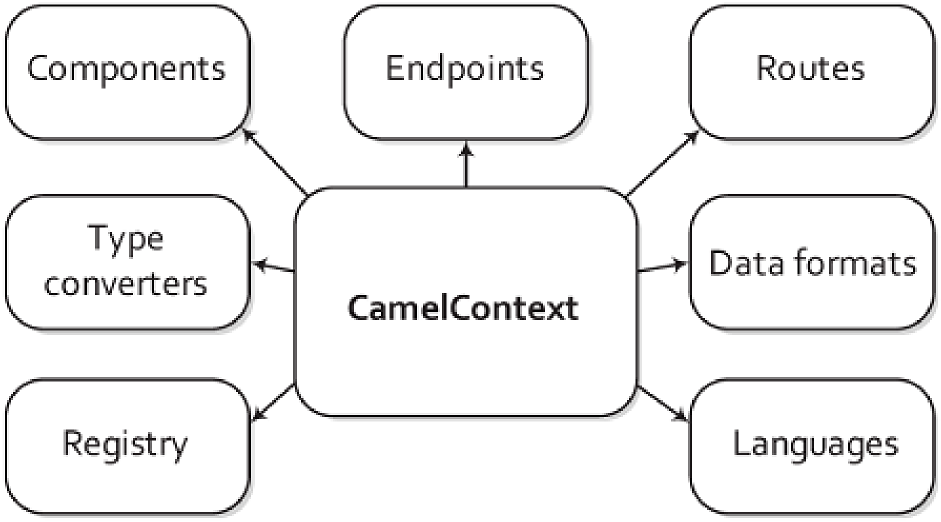


Figure 1.9 The `CamelContext` provides access to many useful services, the most notable being components, type converters, a registry, endpoints, routes, data formats, and languages.

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

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

Table 1.1 The services that the `CamelContext` provides

Service	Description
Components	Contains the components used. Camel is capable of loading components on the fly either by autodiscovery on the classpath or when a new bundle is activated in an OSGi container. In chapter 6 we'll discuss components in more detail.
Endpoints	Contains the endpoints that have been created.
Routes	Contains the routes that have been added. We'll cover routes in chapter 2.
Type converters	Contains the loaded type converters. Camel has a mechanism that allows you to manually or automatically convert from one type to another. Type converters are covered in chapter 3.
Data formats	Contains the loaded data formats. Data formats are covered in chapter 3.
Registry	Contains a registry that allows you to look up beans. By default, this will be a JNDI registry. If you're using Camel from Spring, this will be the Spring `ApplicationContext`. If you are using Camel with CDI, this will be backed by a CDI `BeanManager`. It can also be an OSGi registry if you use Camel in an OSGi container. We'll cover registries in chapter 4.
Languages	Contains the loaded languages. Camel allows you to use many different languages to create expressions. You'll get a glimpse of the XPath language in action when we cover the DSL. A complete reference to Camel's own Simple expression language is available in appendix A.

ROUTING ENGINE

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

ROUTES

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

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

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

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

Under the hood, Camel clones the route definition into 3 separate routes. So, it behaves similar to 3 separate routes like:

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

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

DOMAIN-SPECIFIC LANGUAGE (DSL)

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

Consider this example:

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

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

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

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

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

PROCESSOR

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

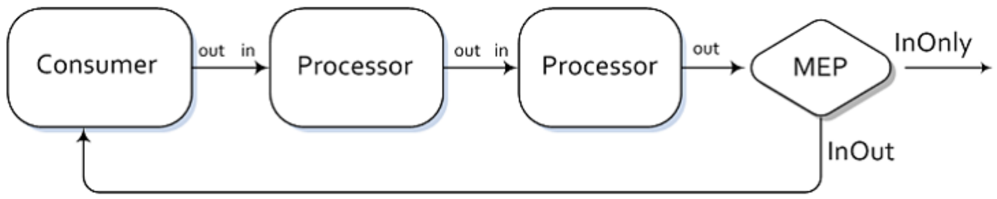


Figure 1.10 Flow of an exchange through a route.

Figure 1.10 shows the flow between processors visually. A route first starts with a consumer (think "from" in the DSL) that populates the initial exchange. At each processor step, the out message from the previous step is the in message of the next. In many cases processors do not actually set an out message so in this case the in message is reused. At the end of a route the MEP of the exchange will determine if a reply needs to be sent back to the caller of the route. If the MEP is InOnly then no reply will be sent back. If it is InOut, then Camel will take the out message from the last step and return it. In this final step the out message **must** be set for a reply to be sent back – the in message will not be reused here.

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

COMPONENT

Components are the main extension point in Camel. To date, there are over 200 components in the Camel ecosystem that range in function from data transports, to DSLs, data formats, and so on. You can even create your own components for Camel—we'll discuss this in chapter 8.

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

ENDPOINT

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

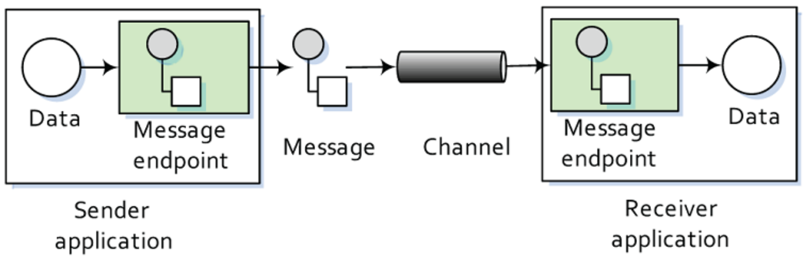


Figure 1.11 An endpoint acts as a neutral interface allowing systems to integrate.

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

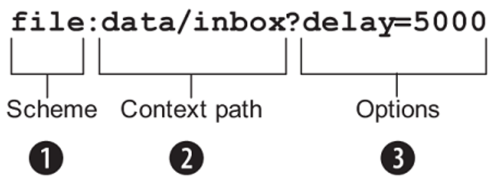


Figure 1.12 Endpoint URIs are divided into three parts: a scheme, a context path, and options.

The scheme 1 denotes which Camel component handles that type of endpoint. In this case, the scheme of `file` selects the `FileComponent`. The `FileComponent` then works as a factory creating the `FileEndpoint` based on the remaining parts of the URI. The context path `data/inbox` 3 tells the `FileComponent` that the starting folder is data/inbox. The option, `delay=5000` 3 indicates that files should be polled at a 5 second interval.

There's more to an endpoint than meets the eye. Figure 1.13 shows how an endpoint works together with an exchange, producers, and consumers.

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

PRODUCER

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

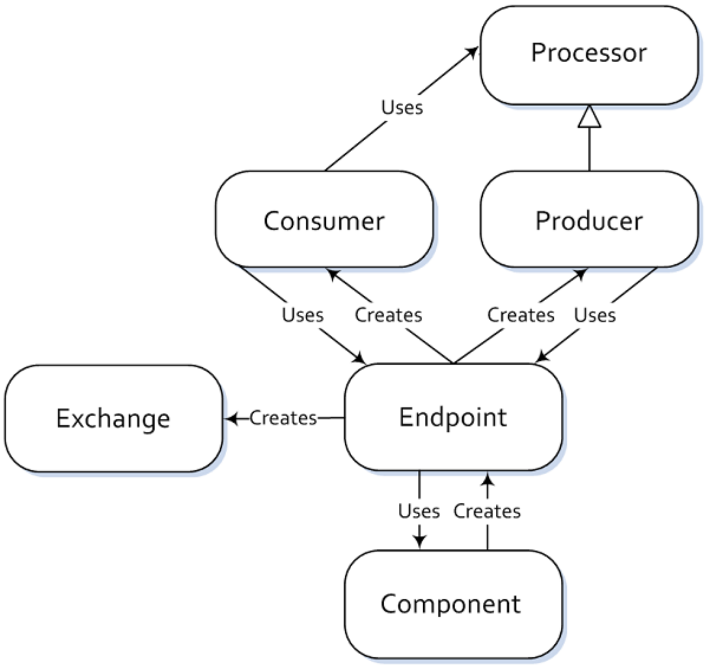


Figure 1.13 How endpoints work with producers, consumers, and an exchange

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

CONSUMER

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

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

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

EVENT-DRIVEN CONSUMER

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

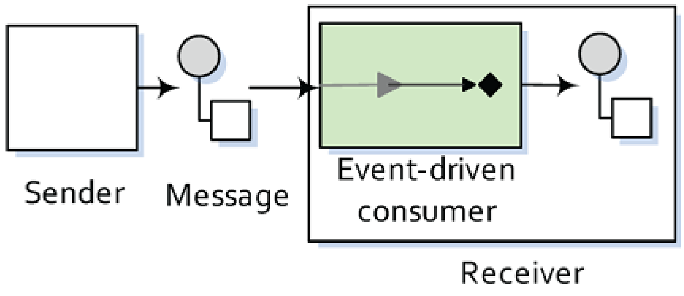


Figure 1.14 An event-driven consumer waits idle until a message arrives, at which point it wakes up and consumes the message.

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

POLLING CONSUMER

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

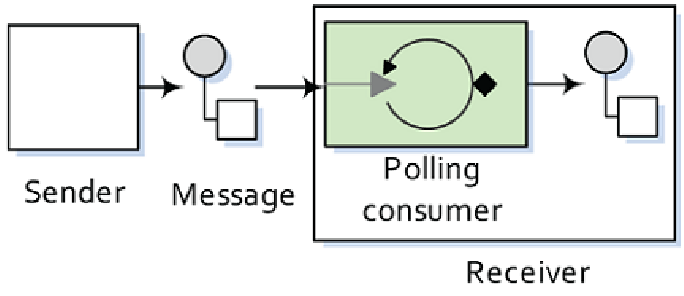


Figure 1.15 A polling consumer actively checks for new messages.

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

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

1.5 Your first Camel ride, revisited

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

Take another look at the Camel application.

Listing 1.4 Routing files from one folder to another with Apache Camel

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

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

(1) **Java DSL route**

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

1.6 Summary

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

- We covered Camel's main features, what Camel is and isn't, and where it can be used. We looked at how Camel provides abstractions and an API that work over a large range of protocols and data formats.
- At this point, you should have a good understanding of what Camel does and what the concepts behind Camel are. Soon you'll be able to confidently browse Camel applications and get a good idea of what they do.
- In the rest of the book, we'll explore Camel's features and give you practical solutions you can apply in everyday integration scenarios. We'll also explain what's going on under Camel's tough skin. To make sure that you get the main concepts from each chapter, from now on we'll present you with a number of best practices and key points in the summary.
- In the next chapter, we'll investigate routing, which is an essential feature and a fun one to learn.