Mastering Apache Camel

This book will provide you with the skills you need to efficiently create routes using Apache Camel.

After briefly introducing the key features and core concepts of Camel, the book will take you through all the important features and components, starting with routing and processors. You will learn how to use beans in Camel routes, covering everything from supported registries and annotations, to the creation of an OSGi bundle and writing route definitions with Blueprint DSL. Leverage the Enterprise Integration Patterns (EIPs) supported by Camel and implement them in your routes. You will then see how components and endpoints handle exchanges in Camel, and how you can use them to create a complete and powerful mediation framework. You will finally learn how to tackle errors and perform testing to ensure that your integration projects are working successfully.

Who this book is written for

This book is intended for all Camel users who want to get the best out of Camel, and who want to implement the most efficient integration logic using best practices.

What you will learn from this book

- Walk through the key features and core concepts of Apache Camel
- Implement routing with Processor, using Camel Java DSL and Camel Blueprint DSL
- Use beans with Camel to access to the POJO model, and leverage IoC frameworks like Spring or Blueprint
- Get to grips with the EIPs supported by Camel and implement them in your projects
- Create polling and event-driven components, and learn how Camel uses components to create endpoints
- Identify and deal with errors in your routing logic
- Conduct unit tests and integration tests on your Camel routes

Jean-Baptiste Onofré

An advanced guide to Enterprise Integration using Apache Camel


Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Core Concepts'
- A synopsis of the book’s content
- More information on Mastering Apache Camel
About the Author

Jean-Baptiste Onofré is a member of the Apache Software Foundation, and he has been involved in Apache projects for about 10 years. He's the PMC chair of Apache Karaf and its subprojects, including Cellar, Cave, and EIK.

He's also a PMC member of Apache ACE, Apache ServiceMix, and Apache Syncope, and he is a committer for Apache ActiveMQ, Apache Archiva, Apache Aries, Apache Camel, and Apache jClouds.

He's currently working at Talend (http://www.talend.com) as a software architect and is a member of the Talend Apache team.

He has provided articles on Java technologies for GNU/Linux magazine France and has worked as an author and a reviewer on different books, such as Learning Karaf Cellar and Apache Karaf Cookbook, both by Packt Publishing.

He has also given talks on Apache projects, such as Karaf and Camel, at different conferences, especially ApacheCon NA and Europe, CamelOne, and so on.
Preface

Apache Camel has slowly emerged as the main framework for integration. It provides a very flexible and efficient way to integrate applications and systems all together.

Camel provides a complete set of features, based on simple but powerful concepts, allowing you to easily implement very rich integration logic.

Using this book, you will have a detailed understanding, with how to steps to implement integration logics.

What this book covers

Chapter 1, Key Features, introduces what Camel is and the provided key features.

Chapter 2, Core Concepts, introduces the basis of all the functionalities provided by Camel.

Chapter 3, Routing and Processors, introduces Camel routing and the usage of processors.

Chapter 4, Beans, explains how to use beans in Camel routes and the different registries in which the beans live.

Chapter 5, Enterprise Integration Patterns, introduces one of the most interesting features of Camel—the ready-to-use patterns, which serve as an answer to classic integration problems.

Chapter 6, Components and Endpoints, introduces Camel components and endpoints, both how to use them and implement your own.

Chapter 7, Error Handling, introduces how to deal with errors in Camel routes.

Chapter 8, Testing, introduces how to implement both unit tests and integration tests on your Camel routes.
This chapter introduces the core concepts of Camel. These concepts are the key basis of all functionalities provided by Camel. We will use them in the next chapters. As we have seen in the previous chapter, Camel is an integration framework. This means that it provides everything to implement your mediation logic: messaging, routing, transformation, and connectivity.

We will look at the following concepts:

- Messages
- Exchanges
- Camel contexts

Messages
Messages transport the data between the different parts of the mediation logic. Your mediation logic will define the flow of messages between different nodes.

A message flows in one direction, from a sender to a receiver. It's not possible to use the same message to answer the sender, we will have to use another message. A message is described in the \texttt{org.apache.camel.Message} interface.


A message contains the following:

- ID: A message ID of type \texttt{String}. Camel creates an ID for you. This ID identifies the message and can be used for correlation or storage. For instance, we will see that the message ID is used in the idempotent consumer pattern to identify the message in a store.
Core Concepts

- **Header**: A set of headers, allowing you to store any kind of data associated with a message. The headers are stored as `org.apache.camel.util.CaseInsensitiveMap` by default. The `CaseInsensitiveMap` extends `HashMap<String, Object>`. This means you can store any kinds of objects (including very large objects) in the header. To access the map use a `String` key, which is case insensitive. The lifetime of the headers is the same as the message (as the headers are part of the message itself). The purpose of the headers is to add hints about the content encoding, authentication information, and so on. As we will see in the next chapters, Camel itself uses and populates the headers for its own needs and configurations.

- **Attachment**: A set of attachments is mostly to match the requirements of some protocols and components: WebService component (to provide SOAP Message Transmission Optimization Mechanism (MTOM) support) or the e-mail component (to provide support for e-mail attachments). The attachments are only used by some dedicated components, they are not as heavily used as the headers. The attachments are stored in the message as `Map<String, DataHandler>`. An attachment name is a `String`, which is case sensitive. An attachment is stored using `DataHandler` providing support of MIME type and consistent access to the data.

- **Fault flag**: A fault flag Boolean that allows you to distinguish whether the message is a normal message or a faulted message. It allows some components or patterns to treat the message in a different way. For instance, instead of a SOAP Response, a message may contain a SOAP Fault. In that case, we have to inform the component that a message containing a SOAP Fault is not a normal message.

- **Body**: The body is the actual payload of the message. The body is stored as an `Object` in the message, allowing you to store any kind of data. In Chapter 1, Key Features we saw that one of the Camel key features is to be payload-agnostic. The fact that the body is directly an `Object` is the implementation of the payload-agnostic feature.
Exchange

Camel doesn't transport a message directly. The main reason is that a message flows only in one direction. When dealing with messaging, there are many Message Exchange Patterns (MEP) that we can use.

Depending on the use cases, we can send a message without expecting any return from the destination: this pattern is named event message and uses InOnlyMEP. For instance, when you read a file from the filesystem, you just process the file content, without returning anything to the endpoint that read the file. In that case, the component responsible for reading the filesystem will define an InOnlyMEP.

On the other hand, you may want to implement a request reply pattern: a response message should be returned to the sender of the request message, and so it uses an InOutMEP. For instance, you receive a SOAP Request from a WebService component, so you should return a SOAP Response (or SOAP Fault) to the message sender.

In Camel, MEP are described in the org.apache.camel.ExchangePattern enumeration (http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/ExchangePattern.html). We can see that Camel supports the following MEP:

- InOnly
- InOptionalOut
- InOut
- OutIn
- OutOptionalIn
Core Concepts

- RobustInOnly
- RobustOutOnly

As a message flows in only one direction, in order to support the different MEPs, we need two messages:

- The first message is mandatory as it's the \textit{in} message
- The second message is optional (depending on the MEP) as it's the \textit{out} message

That's why Camel \textbf{wraps} the messages into an Exchange object: the actual object transported is the Exchange, acting as a messages container with all meta-data required for the routing logic.

This Exchange object is used for the complete mediation process execution.

The \texttt{org.apache.camel.Exchange} interface describes an exchange.

Basically, an exchange contains the following:

- Exchange ID: An exchange ID as a \texttt{String}. This is a unique identifier for the exchange. Camel creates it for you.
- MEP: The Message Exchange Pattern (MEP) defines the exchange pattern.
- Exception: The Exception is used by the error handler, as we will see later. It stores the current cause of an exchange failure. If an error occurs at any time during routing, it will be set in this exception field.
- Properties: The properties is a \texttt{Map<String, Object>} and may look like message headers. The main difference is their lifetime: the properties exist during the whole exchange execution, whereas the headers are limited to the message duration (and a message can change a lot during routing, so during the exchange execution). Camel itself may add some properties for some use cases.
- Finally, we have the \textit{in} and \textit{out} messages.
  
  - In Message: The \textit{in} message is mandatory and always set. It's the only message populated in the exchange with InOnlyMEP.
  - Out Message: The \textit{out} message is optional and is only used with InOutMEP.
With InOutMEP, at the end of the processing of the exchange, the `out` message will be used and returned to the mediation beginner (the first endpoint of the routing who created the exchange).

**Camel context**

The Camel context is the runtime system and the loading container of all resources required for the execution of the routing. It keeps everything together to allow the user to execute the routing logic. When the context starts, it also starts various components and endpoints, and activates the routing rules.

A Camel context contains the following:

- The components and endpoints used in the routing (see later for the details about components and endpoints)
- The type converters used to transform a message of one type to another
- The data formats used to define the format of a message body
- The registry where Camel will look for the beans used in the routing
- The languages describing expressions and predicates used in the routing by a language (xpath, xquery, PHP, and so on)
- The routes definition itself allowing you to design your mediation logic

Most of these resources are automatically loaded by Camel for you; most of the time, as an end user, you specify the routes definitions. However, we will see in the next chapters that we can tweak the Camel context.

A Camel context also has its own life cycle. As it's the runtime system of your routing, you have a control on this life cycle.

A Camel context can be started, loading all resources needed and activating the routing logic.

Once started, a context can be stopped: it's a cold stop. This means that all routes, components, endpoints, and other resources loaded by this context will be stopped, and all internal cache, metrics, and states will be lost.

Instead of stopped, from the started state, a context can be suspended. Suspend stops the routing of the messages, but keeps context resources loaded and the internal data (metrics, cache, states, and so on). That's why a suspended context can return to the started state very quickly using resume: it just resumes the processing of the messages.
The end users are supposed to use only suspend and resume operations.

To restart a context, you can do the following:

- A hot restart using suspend operation first, and resume operation after. It's a fast restart, keeping all the internal data of the context.
- A cold restart using the stop operation first, and the start operation later. In that case, all internal data (cache, states, and metrics) is lost.

Both stop and resume operations will ensure that all inflight messages (messages currently in process) are completely processed.

Stopping or suspending a context will stop or suspend all the routing defined in this context. In order to guarantee a graceful and reliable shutdown of your routing, you can define a shutdown strategy.

A shutdown strategy is described using the org.apache.camel.spi.ShutdownStrategy interface.

Camel provides the org.apache.camel.impl.DefaultShutdownStrategy interface.

This default shutdown strategy works in two phases:

1. First, it does a graceful shutdown, by suspending or stopping all consumer (the first endpoint that creates the exchanges), and waiting for the completion of all inflight messages.
2. After a timeout (5 minutes by default), if there are still some inflight messages, the strategy kills the exchanges, forcing a suspend or stop.

We will see in the next chapters how to create and use our own shutdown strategy.

**Processor**

A processor is a node in the routing which is able to use, create, or modify an incoming exchange. During routing, the exchanges flow from one processor to another. This means all Enterprise Integration Patterns (EIP) are implemented using processors in Camel. The exchanges get in and out of a processor by using components and endpoints, as we will see later in this chapter.
Core Concepts

A processor is described using the `org.apache.camel.Processor` interface. To create your own processor, you just have to implement the `Processor` interface and override the `process()` method:

```java
public class MyProcessor implements Processor {
    public void process(Exchange exchange) {
        System.out.println("Hello " +
        exchange.getIn().getBody(String.class));
    }
}
```

Thanks to the `Exchange` argument of the `process()` method, you have complete access to the exchange: in and out messages, properties, and so on.

The `exchange.getIn()` gets the `in` message of the current exchange. As we want to get the body of this message, we use the `getBody()` method. This method accepts a type argument, casting the body in the destination class (a string in our example).

## Routes

The Camel route is the routing definition. It's a graph of processors. The routes (routing definition) are loaded in the Camel context. The execution and flow of the exchange in a route is performed by the routing engine. The routes are used to decouple clients from servers, and producers from consumers: an exchange consumer doesn't know where the exchange comes from, and on the other hand an exchange producer doesn't know the destination of the exchange. Thanks to that, it provides a flexible way to add extra processing or change the routing with limited impact on the logic.

Each route has a unique identifier that you can specify (or Camel will create one for you). This identifier is used to easily find the route, especially when you want to log, debug, monitor, or manage a route (start or stop).
A route has exactly one input source (the input endpoint). A route has a life cycle similar to the Camel context with the same states: started, stopped, and suspended. To Camel, a context controls the life cycle of the routes that it contains.

**Channels**

In every Camel route, there is a channel that sits between each processor in the route graph. It's responsible for the routing of an Exchange to the next Processor in the graph. The channel acts as a controller that monitors and controls the routing at runtime. It allows Camel to enrich the route with interceptors. For instance, the Camel tracer or the error handling are functionalities implemented using an interceptor on the channel.

The channel is described by the `org.apache.camel.Channel` interface. You can configure your own interceptor on the channels by describing it in Camel context.

Camel supports three kinds of interceptors on the channels:

- **Global interceptors:** This intercepts all exchanges on the channels
- **Interceptors on the incoming exchanges:** This has limited the scope of the interceptor only on the first channel (the one just after the first endpoint)
- **Interceptors on the exchanges going to one specific endpoint:** This limits the interceptor to the channel just before a given endpoint

**Domain Specific Languages (DSL)**

Using the Camel API directly would need you to write a lot of plumbing code. You will need to create all the objects and load a lot of objects into different ones.

Therefore, the direct usage of API would be very time consuming. Moreover, as a flexible and easy-to-use integration framework, Camel doesn't have to force the use of one language (Java) to write the routing logic. Users may not be familiar with Java and might prefer to write their routing logic using another language.

That's why Camel provides a set of languages to directly write the routes: the Camel Domain Specific Languages (DSL).

Using a DSL, the user directly writes their routes and describes the Camel context using a DSL. Camel will load and interpret the DSL to create and instantiate all the objects.
The DSL is used to wire processors and endpoints together to define and form routes. Using a DSL, you mostly define the following:

- The Camel context containing the routing rule base and resources
- The routes definition

Camel supports the following DSLs:

- Java DSL, allowing you to define the routes using a fluent Java API
- Spring XML, allowing you to define the routes using XML and the Spring framework
- Blueprint XML is similar to Spring XML but uses OSGi Blueprint instead of the Spring framework
- REST DSL, allowing you to define the routes using a REST style API (in Java or XML)
- Groovy DSL, allowing you to define the routes using the Groovy language
- Scala DSL, allowing you to define the routes using the Scala language
- Annotation DSL, allowing you to define the routes directly using annotations on Beans

The following routes are exactly the same, but written using two different DSLs.

Using Java DSL:

```java
from("file:/inbox").to("jms:queue:orders")
```

Using Spring or Blueprint DSL:

```xml
<route>
  <from uri="file:/inbox"/>
  <to uri="jms:queue:orders"/>
</route>
```

**Component, endpoint, producer, and consumer**

The components are the main extension points in Camel. We don't directly use a component in a route, we define an endpoint from the component. This means a component acts as a factory for endpoints as follows:
First, you load the component in the Camel context.
Then, in the route definition, you define an endpoint on a component loaded in the Camel context.

You can explicitly instantiate a component and load it in the Camel context (using code), or Camel will try to create and load the component (discover) for you based on the endpoint definition.

Camel provides about 100 components (file, ftp, http, CXF, JMS, and so on) as you can see at http://camel.apache.org/components.html. You can create your own component, as we will see in the next chapters.

Using a component, we create the endpoints. An endpoint represents the end of a channel through which an external system can send or receive messages. It allows your Camel route to communicate with the environment.

Depending on the location in the route, an endpoint can act as follows:

- A producer receives a Camel Exchange, transforms it into an external message and communicates (sends the message) to the external system (environment).
- A consumer receives a message from an external system (environment), wraps it as a Camel Exchange, and sends to the route.

We identify two kinds of consumers:

- Event Driven Consumers who will listen for incoming messages and create a Camel exchange at this time. For instance, consumer endpoints using the CXF component will react when they receive a SOAP Request.
- Polling Consumers who periodically check for new resources and create a Camel exchange at this time. For instance, consumer endpoints using the File component will periodically poll the filesystem and create a Camel exchange for the new files.
Core Concepts

The endpoints are described using a URI in the following format:

    component:option?option=value&option=value

For instance, we can define an endpoint of the file component using the following code:

    file:data/inbox?delay=5000&noop=true

At runtime, Camel will look up an endpoint based on the URI, check whether the component defined as prefix is in the Camel context (eventually load it, if it's not there), and use this component to actually create the endpoint.

Data format

Camel supports a pluggable data format allowing you to marshall and unmarshall the messages.

For instance, if you receive an XML message from an endpoint, you can:

- Directly manipulate and transport the XML message in the route
- Translate the XML to a POJO using JAXB, for instance, JAXB is a data format

Unmarshalling allows you to use a data format to convert from a raw format (XML in the previous example) into a Java object. On the other hand, when you send an exchange to an endpoint, you can marshall the transported object into another format. You specify where you want to unmarshall or marshall and the data format to use.

For instance, you can consume an XML message from a JMS queue, unmarshall using JAXB, and send the resulting object to another JMS queue:

    from("jms:queue:orders").unmarshal("jaxb").to("jms:queue:other")

You can also unmarshall files containing a serialized object, and then marshall using JAXB to send the resulting XML message to a JMS queue:

    from("file:data/inbox").unmarshal().serialization().marshall("jaxb").to("jms:queue:orders")

Camel provides a lot of ready-to-use data formats:

- For JVM native serialization marshalling, you can use the serialization or Stringdata formats
- For object marshalling, you can use Avro, JSON, or Protobuf data formats
• For XML data formats (marshalling and unmarshalling), you can use JAXB, XmlBeans, XStream, JiBX, or Castor libraries
• For XML or WebService marshalling, you can use the SOAP data format
• For XML or JSON marshalling, you can use the XmlJson data format
• For flat data structure marshalling (CSV, DelimitedLength, and so on), you can use BeanIO, Bindy, CSV, EDI, or Flatpack data formats
• For compression marshalling, you can use GZip or Zip data formats
• For security marshalling, you can use PGP, Crypto, or XML Sec data formats
• For other marshalling, you can use Base64, RSS, TidyMarkup (with HTML, for instance), or Syslog data formats

You can also create your own data format, providing the custom marshalling and unmarshalling mechanism.

**Type converter**

Even without a data format, when you route a message from one endpoint to another, it's common to convert the body of the message from one type to another. For instance, in an exchange created by a file endpoint, the body of the in message will be an InputStream. Depending on the target endpoint or processor, we may want to convert this InputStream to a String.

When you use the `getBody()` method on a message, you can specify the expected type. Camel will use a type converter to try to convert the actual body of the message into the specified type.

For instance, in a processor, if you do the following:

```java
Message in = exchange.getIn();
Document document = in.getBody(Document.class);
```

Camel will try to convert the body of the in message into a DOM document. A type converter is defined by the `org.apache.camel.TypeConverter` interface. The type converters are loaded into the Camel context, in a type converter's registry.

The type converter's registry contains the type converter with the supported types. In this registry, for each type converter, we have:

- The **source** type
- The **destination** type
- The actual type converter instance
Core Concepts

For instance, we can add our own type converter in the Camel context as follows:

```java
context.getTypeConverterRegistry().addTypeConverter(MyOrder.class,
String.class, new MyOrderTypeConverter());
```

We can see that the source type is `MyOrder`, the destination type is `String`, and to convert from a type `MyOrder` to `String`, I will use the `MyOrderTypeConverter()` method.

Summary

We can see that even if the Camel core is light, the provided features are rich and it provides all the basics to extend Camel to match your needs.

This chapter is an introduction to the Camel core concepts. It allows you to easily understand the next chapters, where we will get into the details of Camel.
Where to buy this book

You can buy Mastering Apache Camel from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.