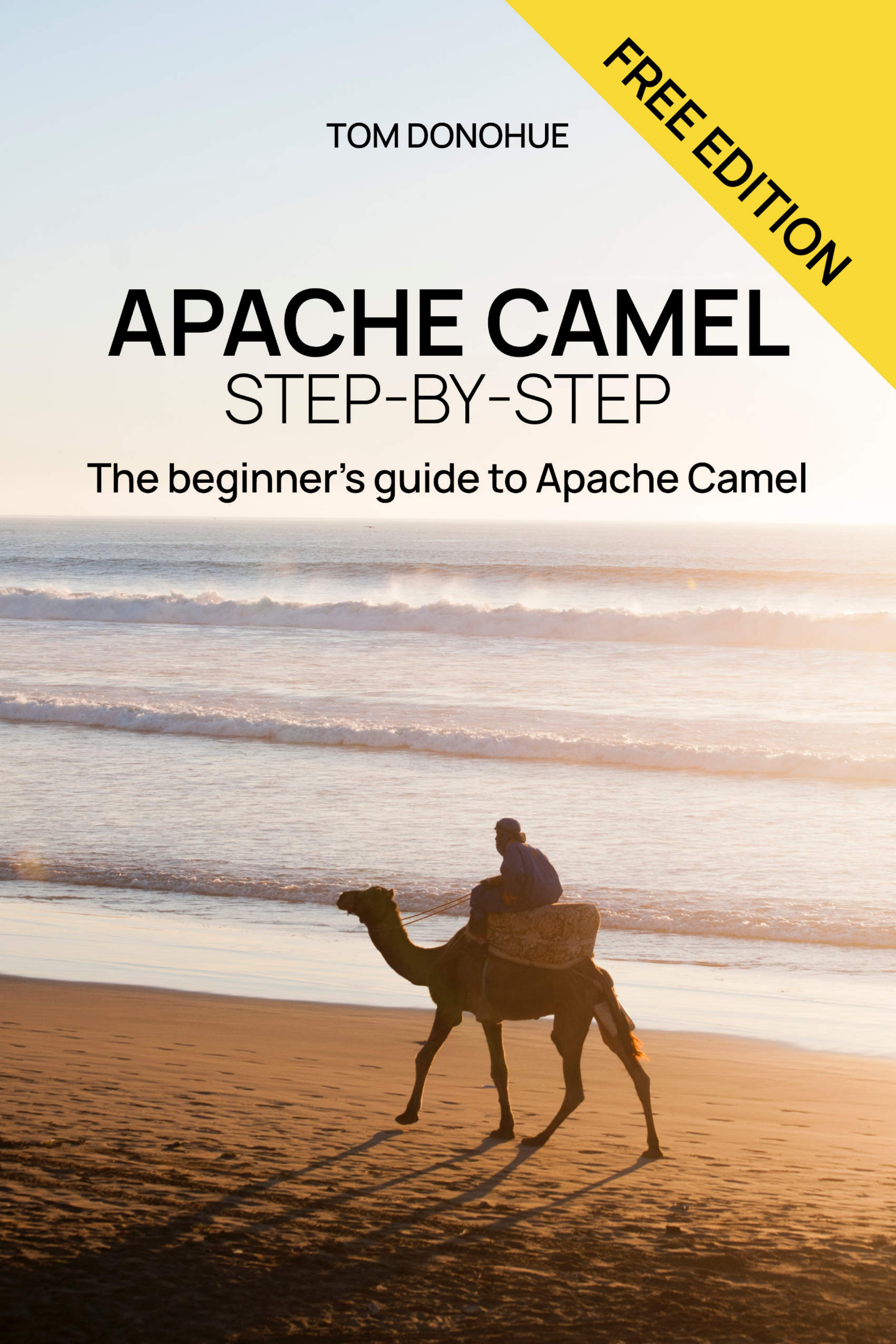


FREE EDITION

TOM DONOHUE

APACHE CAMEL STEP-BY-STEP

The beginner's guide to Apache Camel

Apache Camel Step-by-Step - Free Edition

Tom Donohue

Table of Contents

Preface

Hey! Hello! Hi!

Thanks very much for purchasing this book.

When I first discovered Apache Camel, I was working for a finance company in London on an integration project. The company was using an integration product called Talend, which uses Apache Camel as its integration layer.

Working with Apache Camel was like a breath of fresh air. I liked it for its elegant way of solving integration problems, and the fact that it is an open-source Java library so I can see the source code to understand what's going on.

I started to write some articles on Camel (the ones you now see on my website, tomd.xyz). I wrote about common tasks that you might want to do in Camel and core concepts that might help new developers.

Those articles grew in popularity over the last couple of years, and so has Apache Camel. There are new developers learning Camel every day just like yourself, so you're in very good company. (See the section on community at the back of the book).

This book is designed to help you learn the basics of Camel, and get started with your first applications. I hope you enjoy this book, and that you enjoy learning Apache Camel. I also hope that you will get truly hooked, like I did.

Cheers, and happy Camel riding!

Tom Donohue

Introduction to this book

So, you want to get started with Apache Camel the easy way.

Maybe you've tried other tutorials, but they all seem too complicated. Or half-finished. Or unreadable. Or just a bunch of code pasted onto a web page with some annoying ads.

If you're the kind of person who prefers to start learning a technology with a step-by-step approach, then here's some good news: I wrote this book for you.

To begin, I'm going to **introduce Apache Camel**. You'll learn the basics about what Camel is, and what it is designed to do. You'll understand what you need to know as a new Camel developer. Then, you'll learn about basic Camel concepts, like routes and components, and how they all fit together in Camel's architecture.

Then, you'll **create your first Camel application**. We'll generate a new Camel app from a template, look at the source code, run it from the command line.

Getting ready

To get the most out of this book, it's best to follow the tutorials, step-by-step. To help you do this, you'll need the following tools installed on your computer:

Java Development Kit (JDK) 11 - this is your toolkit for building Java applications. **(search online for "java development kit" and download the "Java SE JDK")**. If you're using a Linux-based system, try using **SDKman** to install Java.

Apache Maven - Maven is a build helper for Java. It helps you build your applications, and manages all of your dependencies for you.

You also need to be working on a computer which is **connected to the internet**, so that Maven can download the Camel dependencies when it needs to.

Updated for Camel 3

This book has been updated to work with Camel 3, the current major version of Camel.

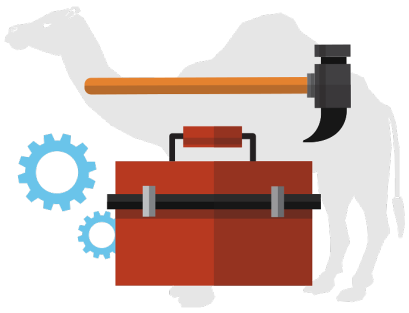
The architecture of Camel has changed slightly in version 3. Camel introduced some changes, to make it more modular, and lightweight for running in the cloud. This means that Camel 3.x may be slightly different to a version of Camel that you have used before.

Camel basics

> What exactly is Apache Camel?
>
> What do Camel routes look like?
>
> How do messages work in Camel?
>
> How do you create a simple Camel project?

In this part you'll learn the very basics of Camel.

Introducing Apache Camel



OK, let's start with the BIG question. Lots of people start their journey with Apache Camel by asking this question:

What exactly is Apache Camel?

Apache Camel is an **integration library** for Java. It provides a set of **Java APIs** which help you integrate and process data between different computer systems. In other words, Camel is like the glue between different applications.

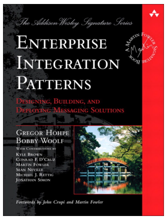
You can think of Camel like a **plumbing toolkit for your data**. Camel can take data from one application, and pipe it to another. Along the way, the data can be changed, transformed, or sent through other pipes.

Included in your plumbing toolkit is a range of adaptors (these are called **Components** in Camel-speak) which fit all kinds of applications and software systems.

Components in Camel help you to integrate data from lots of different places — like data in web services, or data in files on disk. You can even connect Camel to web apps, like Twitter and Salesforce.

With your Camel toolkit at your disposal, it's up to you how you choose to build the plumbing.

Where did the idea for Camel come from?



Camel was heavily inspired by the book **Enterprise Integration Patterns**, written by Gregor Hohpe and Bobby Woolf. It’s a textbook which proposed some common patterns for integrating software systems.

The authors of the book (now considered a 'classic' in some circles!) looked at the different ways that people were integrating software systems, and distilled them into reusable patterns.

For each pattern, they wrote a little bit about it, described how it might be used, and provided some suggested code for how to implement the pattern.

The original Camel developers thought it would be a great idea to build a Java framework which actually implemented the patterns described in the book.

And so, many of Camel’s core features are based heavily on the book. In fact, Camel provides things called **Enterprise Integration Patterns (EIPs)**, which are named directly after the patterns in the book. You’ll learn about these later on.

Camel in 5 points

Here are some high-level pieces of information… and reasons to **befriend a Camel!**

Camel is written in Java… Since Apache Camel is written in Java, you have the full power of Java at your disposal. You can embed it in your existing applications, integrate Camel with your own Java code, and much more.

The source code is completely open… Want to know exactly how Camel works? Just grab the Camel source code from GitHub and have a look around. Everything is there. Nothing is hidden away in proprietary code.

It’s not just for web services… it’s a general integration framework. You can use Camel to interact with web services, but it’s capable of far more than that.

It comes with a huge library of components… if you can think of an application or service you’d like to connect to, there’s probably a Camel component for it. You can do everything from storing files in Dropbox, to sending a tweet. You can also write your own component, and contribute it to the community.

It’s mature… Apache Camel is well and truly mature, and has a great community behind it. It also forms the foundation of some commercial integration products, like **Red Hat Fuse** and Talend ESB. This means that you can always get support for Camel if you need it, whether paid commercial support, or community help.

What is Apache Camel used for?

Whenever you need to move data from **A** to **B**, you can probably use Apache Camel.

Here’s some scenarios, to give you an idea of what you can build with Camel. You can implement **all** of these with Camel — usually with just a few lines of code:

- Pick up invoices from your corporate file server and email them to customers
- Move documents from your hard disk into Google Drive
- Take incoming customer order messages from a queue, and use them to invoke a web service
- Turn some XML documents into CSV files

- Make a web service which serves product information from a database

These are just a few examples. With the huge number of components available for Camel, the sky’s the limit.

Plus, Camel is already used by hundreds of companies and organisations. Banks, insurance companies, retail stores and governments all use Camel to integrate and move data between computer systems.

So you’re in good company.

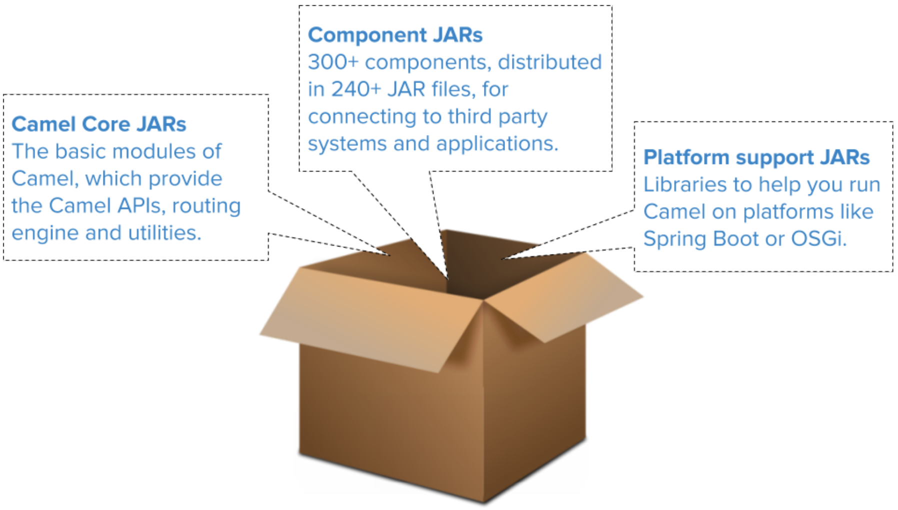
Now we’ll take a look inside the Camel box, and see how it works.

What's in the box?

Apache Camel is not a program that you install. It's a library for Java developers, distributed as a set of JAR files that you can include in your own projects.

Most developers will get started with Camel by adding it as a dependency using Maven, and we'll see how to do this shortly.

But if you were to download the binary distribution of Camel from the Camel web site, what would be in the box?


Component JARs
300+ components, distributed in 240+ JAR files, for connecting to third party systems and applications.
Camel Core JARs
The basic modules of Camel, which provide the Camel APIs, routing engine and utilities.
Platform support JARs
Libraries to help you run Camel on platforms like Spring Boot or OSGi.

Camel Core

The core functionality of Camel is distributed as a set of about 12 JAR files. This includes the Camel APIs, the Camel runtime engine, and functionality for runtime management of your Camel applications.

Component JARs

To allow Camel to connect to external systems, Camel needs to be given super-powers, by adding **components**. Each component extends Camel with functionality to interact with a third party system or API. Most components are distributed in their own JAR. There are over 300 components to choose from.

Platform support JARs

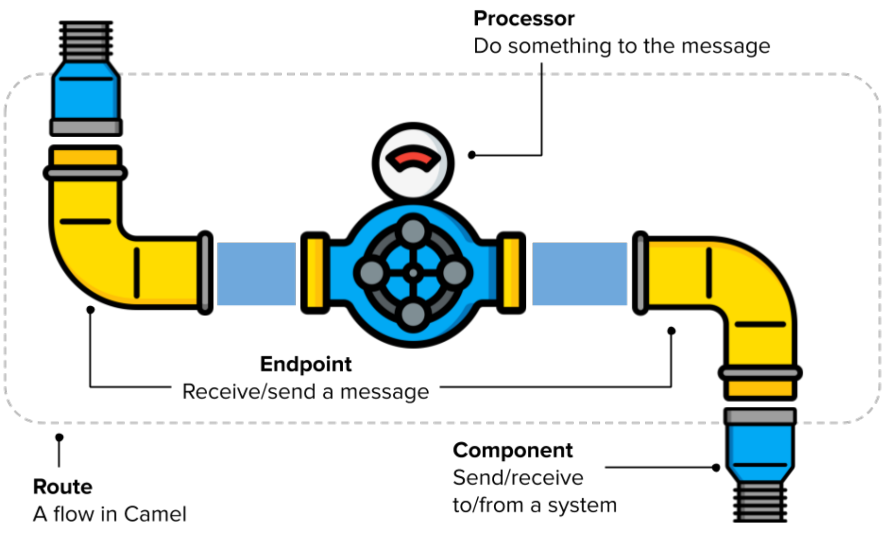
> These JAR files provide functionality to help Camel run on different platforms, like Spring Boot, or Apache Karaf (OSGi).

Camel is designed to be added to your existing Java application. Or, you can start with Camel by creating a new project, using one of the templates provided by the Camel community.

> Versions of Camel before 3.x included more core functionality in the Camel Core JAR. However from Camel 3.x, the Camel Core JAR has been slimmed down, and many components and features have been moved out into their own, dedicated JARs.

Camel concepts

How do you ride a Camel? In this section we'll look at each of the main concepts of Apache Camel.



Camel Context

Camel provides a runtime engine, called the **Camel Context**, which runs your integrations.

It's kind of like a mini application server, embedded inside your Java application.

When you work with Camel, you define your set of integration pipes (in either Java or XML syntax), add them to the Camel engine (the Camel Context), and start the engine. Then, you can leave the engine running, and it will continue running your integrations until you switch it off.

The Camel Context runs your integrations and is generally responsible for moving data through the system.

The simplest way of starting Camel is to create an instance of

`DefaultCamelContext` (which implements the `CamelContext` interface) and call the `start` method:

Listing 1. Starting the CamelContext

```
CamelContext context = new DefaultCamelContext();
context.start();
```

But wait! The Camel Context is very little use to us without some integrations running inside it. So let's cover those next.

Route

One of the main reasons to use Camel is when you want to integrate systems and applications together. To use the same plumbing analogy from earlier in this section, we want to use Camel so that we can run pipelines which move data from one place to another.

In Camel, these integration pipelines are called **Routes**.

A route usually consists of a series of steps. The first step receives or fetches some data, and then the data is passed through subsequent steps to the end of the route.

You don't run a route directly. Routes are configuration objects defined in Camel using a Java or XML syntax, and then they are added to the Camel Context.

A simple route definition looks something like this (in Camel's Java syntax):

Listing 2. A simple Camel route

```
from("file:documents/invoices") ①
        .to("file:documents/done"); ②
```

① The first, receiving step in the route

② The second and final step in the route

We'll cover more about routes in the next section.

Endpoint

When writing integration routes in Camel, you will often need to interact with other systems.

In Camel, an **Endpoint** is an interface through which Camel exchanges a message with another system. Camel can receive a message from an Endpoint, or send a message to an Endpoint.

Endpoints are the physical representations of the external applications in your routes, and you declare them using a URI.

For an example, at the start of a route, Camel will **receive** a message from an endpoint. As an example, this endpoint might be a `file:` endpoint, which reads a file on disk.

You just saw two examples of using a `file:` endpoint above.

Then, the message might be processed or transformed in some way before being sent to another destination endpoint.

An endpoint in a Camel route is generated by a **Component**. Components, as you'll see next, are Camel's connectors that allow it to interact with external systems and applications.

Component

To allow Camel to create an **Endpoint**, it uses a **Component**. A Component is simply a specialised plug that allows you to connect to an external system, such as a file on disk, a mailbox, or an application like Dropbox or Twitter.

You can think of a Component as a factory for creating Endpoints.

Whenever you want to move data in or out of an application, you'll probably find that a Camel Component already exists to do the job for you.

This means you don't need to waste time writing your own code to read a file, or invoke a web service. Just find the Component you need, and use it.

Components are reusable, open source, and you can even create your own.

To give you a better idea of Components, here are some common ones, and how you might reference them in an **Endpoint**:

Table 1. Common components in Apache Camel

Component	Purpose	Endpoint URI
File	read or write a file	`file:`
JMS	read or write to message queues	`jms:`
Direct	join your Camel routes together	`direct:`
HTTP	make an HTTP request	`http:`

Each component can usually read and write to the target system:

- A component that is configured to **write something** is called a **producer** — for example, writing to a file on disk, or sending to a message queue.
- A component that is configured to **read something** is called a **consumer** — for example, reading a file from disk, or perhaps receiving a REST request. We usually start a route with a consumer.

How do you know which components are available for Camel?

Camel ships with a **lot** of components. The Camel web site has documentation for all of the components included with the current version of Camel.

To see the current component list, and read the documentation pages, visit:

https://camel.apache.org/components/latest/

There is also a small number of extra components which, for licensing reasons, can't be included in the Camel distribution. These can be found in the **Camel-Extra** project:

https://camel-extra.github.io/

Processor

Processors are another step in your Camel routes. They allow you to run some custom code against your messages.

A processor is usually a Java class that implements camel's `Processor` interface.

You might use a processor to:

- Perform your own custom actions that can't be performed using Camel features
- Implement some business logic
- Call out to another system or Java API

Processors, and running custom Java code in general, are covered in the full edition of this book.

Enterprise Integration Patterns

Enterprise Integration Patterns, or EIPs, are another important part of Camel.

In Camel, EIPs are Java-based implementations of the patterns which are described in Gregor Hohpe and Bobby Woolf's book.

Some EIPs are a fundamental part of Camel, and you will use them without even realising it. One example, the Pipes and Filters pattern, is represented as a **Route** in Camel.

Other EIPs may appear more explicitly in your routes, such as the **Loop** EIP. You can use the Loop EIP to repeat some steps in a route.

Some other examples of explicitly-invoked EIPs are in the table below:

Table 2. Common EIPs in Apache Camel

EIP name	What it does	Java syntax in Camel
Splitter	Splits a message into multiple parts	`.split()`
Log	Writes a simple log message	`.log()`
Marshal	Converts an object into a text or binary format (sometimes called **serialisation**)	`.marshal()`

There are lots of EIPs, and over time you'll get to know most of them. For now, you don't need to worry about them all.

If you want to see the complete list of patterns implemented in Camel, you can view them here: View all the Camel EIPs. You can also view the Enterprise Integration Patterns website for more info.

Writing your routes

So now we've learned that to develop in Camel, you create **routes** that move data between **endpoints**, using **components**.

This is all still quite theoretical at this stage. It's probably easier to understand all of this by looking at some code, right?

Routes are code

I said in the previous chapter that routes are configuration objects which are applied to a Camel Context.

In practice, this means that we first define a route in code, and then attach it to a Camel Context.

Although Camel is a Java-based framework, it can be configured using either Java or XML syntax. In Camel-speak, the syntax you use is called a **DSL** (Domain Specific Language).

- **Java DSL** - this uses Java syntax for defining routes. You write your routes in a `RouteBuilder` class and use a "fluent" builder, which can be used to chain multiple route steps together.
- **XML DSL** (also sometimes called **Spring DSL** or **Blueprint DSL**) - this is where you declare your routes in an XML file. A route is an XML element which contains multiple child elements, which represent your steps.

The syntax of each DSL is different, but the overall structure of a route is mostly the same, whether it's written in the XML DSL or the Java DSL.

Which DSL should you use?

If you're starting a brand new Camel project, you're probably going to ask which DSL you should use. The answer to this is: it really depends!

I prefer the Java DSL, because it's manifested as regular Java code, which means that I can use all of the language constructs that I'm familiar with in Java. Java configuration is also preferred by Spring Boot over XML.

However, the XML DSL is more readable for some people, and might be ideal for you if you're used to XML-based configuration of your application. This might certainly be the case if you're familiar with Spring.

Want to know more? Bilgin Ibryam wrote a good blog post on this topic, which you can find here.

A basic Camel route

Each route starts with a `from` Endpoint, usually specified as a URI, which defines where the data should be received from.

A route can consist of **multiple steps**: such as transforming the data, or logging it. But a route usually ends with a `to` instruction, which describes where the data will be delivered **to** next.

A really simple route in the Java DSL could look something like this:

Listing 3. A Java DSL route that moves files

```
from("file:home/customers/new") ①
        .to("file:home/customers/old"); ②
```

① Pick up files from one folder....

② And move them to another folder

In this example, we use the **File** component (identified by the `file:` prefix) to move all incoming files in the `home/customers/new` folder, to the `home/customers/old` folder.

This same route could be expressed in the XML DSL like this:

Listing 4. An XML DSL route that moves files

```
<route>
    <from uri="file:home/customers/new"/>
    <to uri="file:home/customers/old"/>
</route>
```

What if we want to add another step in our route? We want to log a message when we've received a file. Now we simply add a new step, **between** the existing steps, like this:

Listing 5. A route that writes a log

```
from("file:home/customers/new")
        .log("Received a new customer!") ①
        .to("file:home/customers/old");
```

① This step writes a log message to the console

Now, in the code above, a message is moved from the `new` folder to the `old` folder. In the middle, we write a log message to the console, using the **log** step.

And in the XML DSL, it would look something like this:

Listing 6. An XML DSL route that writes a log

```
<route>
    <from uri="file:home/customers/new"/>
    <log message="Received a new customer!"/>
    <to uri="file:home/customers/old"/>
</route>
```

Routes can be more complex than this, of course, but what we have written is a fully-functioning route. In only three lines, Camel can do things that you would normally have to write a lot more Java code for.

What happens at the end of a route?

At the end of a route, Camel will do one of two things, depending on a property of the message, known as the **Message Exchange Pattern**, or MEP.

Return the final output message back to the consumer

If the route has been invoked in a synchronous way, then the Message Exchange Pattern is set to `InOut`. This happens when the original caller expects some kind of response.

In this case, Camel will return the output message from the final endpoint back to the caller, and complete the route.

A good example of this is Camel's **Jetty** component, which configures an embedded Jetty web server. When you use the Jetty component as the consumer at the start of a route, it will listen for HTTP requests. When an HTTP request is sent by a client, the Jetty endpoint receives it and sets the Message Exchange Pattern to `InOut`.

In this example, a Jetty web server listens on localhost port 8008. When a request is received, the response `Hi!` is returned back to the consumer, even though there is no explicit code configuring it to do so:

Listing 7. An InOut MEP: returning a response to an HTTP request

```
from("jetty:http://localhost:8008/hello") ①
        .transform(constant("Hi!")); ②
```

① Receive an HTTP request at `/hello`

② Set the Body to `Hi!`

Do nothing

If the component at the start of the route is a consumer of a **one-way** message — in other words, if the consumer received a message from a caller that doesn't expect a response — then the Exchange pattern is set as `InOnly`. In this case, Camel will simply process the final step in the route and finish.

An example of this is the File component. A route which starts with a standard File component will consume a message from disk and not attempt to return a response. There is nowhere to return a response to, so it's considered an `InOnly` pattern.

Listing 8. A File consumer typically produces an InOnly MEP

```
from("file:documents/invoices") ①
        .to("file:documents/done"); ②
```

① Receive a file from a directory

② Move that file to another directory

Messages in Camel

Camel works with data using a message model.

What does this mean? It means that Camel treats data in your routes as **messages** – like letters flowing through a post office.

Each message is an individual unit. A message may be large, or very small. Routes may be processing messages like:

- A web service request payload
- A file from disk
- A message received from a JMS queue

Camel has an API object to represent a message. It's called `Message`.

A `Message` has a **body**, where the message content lives. It also has **headers** and **properties**, which can be used to hold values associated with the message. The `Message` object is then passed along a route.

Introducing the Exchange

A `Message` is part of a bigger Camel object called an `Exchange`. You'll often see the term `Exchange` mentioned in the Camel documentation. An Exchange is simply a message or interaction currently taking place inside your Camel route.

An Exchange is created at the start of a route (from your first `from` step). It is updated as a message flows through the steps of the route.

The **output** message from the first step in the route becomes the **input** message to the second step. This process then keeps repeating until the end of the route.

At any given point in a route, the Exchange object holds the current **request** and **response** Messages.

What’s in the exchange?

Sometimes, when working with Camel, in addition to writing code in routes, you might also want to modify the Exchange object directly. This can be done using a custom Java class, which we’ll cover later in the book.

But what would the Exchange object look like? Here’s an example of how you might interact with these objects in Java code. Here, the `exchange` object is a reference to the current Camel Exchange.

To directly access the `Exchange` object in Java code like this, we can use a **Processor** in Camel. Processors are covered in the full edition of this book.

The exchange has a **body** and a **header** value, which we can fetch from Camel’s `Exchange` object:

Listing 9. Extracting the message Body and Headers from the Exchange

```
Message message = exchange.getMessage(); ①

String header = message.getHeader("MyHeader", String.class); ②

String body = message.getBody(String.class); ③
```

① Get the Message from the Exchange

② Get a Header called 'MyHeader' from the Message, as a String

③ Get the Body of the Message, as a String object

For more information, check out the Javadoc for the `Exchange` object

What can a Message contain?

The example above assumes that the message **Body** is a String. But Camel's message body can contain almost **any kind of Java object** — it doesn't have to be XML, JSON or even plain text for that matter. It can be a Java collection, a POJO, or a complex object.

The real **power** of Camel will become clear to you when you start using different types of objects. Camel has built-in support for converting between many different object types. In fact, for many common object types, you might not need to write any code to convert between them.

Summary of the Exchange

Here's a summary of the Exchange in Camel. You will use these objects a lot when working with Camel:

Table 3. Parts of the Exchange in Camel

Component	What it is
Message	Represents an inbound or outbound message which is created during the execution of a Camel route.
Headers	A set of key-value pairs, which define properties of the `Message`, or any custom values you want to store.
Body	The place where the content of your `Message` is stored - e.g. an email body, or an HTTP request body.
Exchange	A container for the Message object. It holds the current **input** and **output** messages, and other metadata about the current request.

Components and endpoints

Endpoints in Camel are the **building blocks** of your routes.

They are the pieces in your pipeline that receive messages, send messages, and they are backed by **components**, which contain the logic to connect to all sorts of external systems.

There are components in Camel for things like:

- file handling
- messaging
- web services
- cloud apps (like Salesforce and Twitter)
- special utilities (like the Timer component)

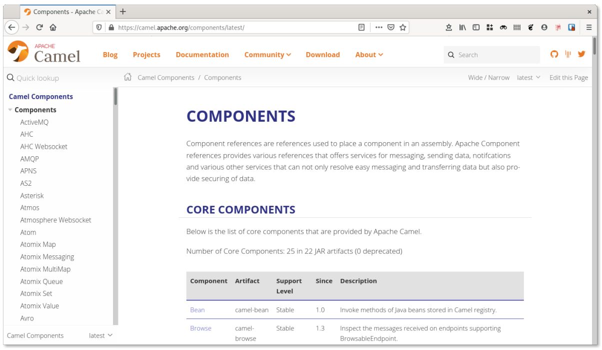
There are over 300 components for Camel, and the number gets bigger all the time.

When you learn Camel, you won't possibly know all of the components at the start. This is OK! Instead, you'll discover new components as you need to use them.

Finding components

Want to find out about all the Camel components available?

The first port of call is the Camel website, where there is a page which lists all of the components available. This is a good starting point when you want to find out if Camel has a component you need:


COMPONENTS

Component references are references used to place a component in an assembly. Apache Component references provides various references that offers services for messaging, sending data, notifcations and various other services that can not only resolve easy messaging and transferring data but also provide securing of data.

CORE COMPONENTS

Below is the list of core components that are provided by Apache Camel.

Number of Core Components: 25 in 22 JAR artifacts (0 deprecated)

Component	Artifact	Support Level	Since	Description
Bean	camel-bean	Stable	1.0	Invoke methods of Java beans stored in Camel registry.
Browse	camel-browse	Stable	1.3	Inspect the messages received on endpoints supporting BrowsableEndpoint.

You can also contribute your own components. Just see the Camel documentation.

How to use a component

When you decide you want to use a component in Camel, you generally need to do two things:

1. Add the appropriate dependency to your Maven `pom.xml`. For most components in Camel, you'll need to include the component as a dependency. The component's documentation page will tell you what Maven dependency you need to add.

   For example, to use the CSV component, which gives Camel the ability to work with CSV files, add the `camel-csv` dependency to your Maven POM:

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

2. Then, to use the component in your routes, add a `from` step (if you want to consume from an endpoint of this component) or `to` step (if you want to send a message to the endpoint).

   On the step itself, set the URI property to your component's **Endpoint URI**, which is a way of telling the component to create a new endpoint with your desired configuration.

Defining an endpoint URI

A component **endpoint URI** usually looks like this:

```
scheme:mainpart?myParam=XXX&anotherParam=YYY
```

- The `scheme` at the start indicates which component we want to use, e.g. `file:`.
- The `mainpart` section usually specifies the location, destination or main argument that the component should use; for example, when using the File component, the main part specifies the **folder** where we want to read or write a file.
- The `optionN` parameters at the end are used to provide more configuration to the component.

The documentation page for each Camel component shows the endpoint URI format you should use.

Example: Using the File component

This is an endpoint URI to configure the File component:

```
file:myfiles/input?fileName=MYFILE.TXT
```

Where:

- `file:` specifies that we want to use Camel's File component
- `myfiles/input` is the folder that we want to read from **OR** write to
- `fileName=MYFILE.TXT` is an **option** we've added to the File component, to tell it to consume only files named `MYFILE.TXT`

Now we can use the endpoint in a route. In the example route below, we're using two different File endpoints in the same route:

Listing 10. Route with two File endpoints (Java DSL)

```
from("file:myfiles/input?fileName=MYFILE.TXT")
        .log("I'm doing something")
        .to("file:documents/done");
```

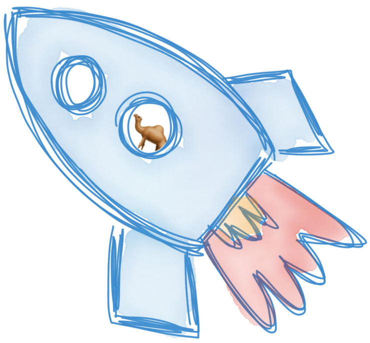
And the same route in the XML DSL:

Listing 11. Route with two File endpoints (XML DSL)

```
<route>
    <from uri="file:myfiles/input?fileName=MYFILE.TXT"/>
    <to uri="file:documents/done"/>
</route>
```

You’ve learned the basics - we’ll pick up the rest on the way. Let’s now put things into practice and walk through creating your first Camel project. Head to the next page to get started!

Your first Camel project



Now that you've learned about the basics of Apache Camel, let's launch straight in with your first project.

In this chapter, you'll create a simple Camel application which moves files from one folder to another.

We'll go through the same steps I use when creating a new Camel app. We'll generate some boilerplate code, run the application, and you'll begin to see how Camel works.

For this first project, we'll be using Spring Boot as the **runtime** for our Camel application.

Why Spring Boot as a runtime?

Spring Boot is an extremely popular framework for Java. It's an **app-in-a-box**, ready for you to start coding, using Spring's **Dependency Injection** style.

I've chosen Spring Boot for this tutorial because it gets you started quickly with Camel, and there is a huge community of Spring Boot users. You've may have already used it yourself.

Teaching Spring is beyond the scope of this book, but you can start learning Spring with the guides at spring.io.

So what are we going to do?

- Create a new Camel on Spring Boot project, using a Maven **archetype**
- Inspect the source code
- Compile it, run it and see what happens!

The Maven Archetypes for Apache Camel are a very good starting point when you're creating a new project. Each archetype usually contains at least one or two sample Camel routes to get you started, and the archetype collection is updated with every release of Camel (e.g. 3.4.2).

So when you use an archetype, you basically get a fully-working Camel application, which follows best practices, which you can replace with your own code. Nice!

Creating the new project from a Maven archetype

You'll need to be connected to the Internet to complete this section.

Now let's create a basic Camel application, using Maven at the command line.

To create a project from an archetype, we run Maven with the command `mvn archetype:generate`.

Archetypes are published by open source communities into different **groups**. Camel archetypes, for example, are published to the group `org.apache.camel.archetypes`.

Let's now create a new project from the **Spring Boot** archetype. Use this command in your terminal or command prompt. Note I'm using backslashes `\` to split this command over multiple lines. If you're using Windows, then replace them with underscores `_` instead:

```
mvn archetype:generate \
   -DarchetypeGroupId=org.apache.camel.archetypes \
   -DarchetypeArtifactId=camel-archetype-spring-boot \
   -DarchetypeVersion=3.4.2 \
   -DgroupId=com.example \
   -DartifactId=first-app \
   -Dversion=1.0-SNAPSHOT
```

You might need to press **Y** (Yes) to confirm the new project, when prompted.

When this command completes, you should have a new folder, `first-app`. Read on to find out what's inside.

We've just created a Camel project from the Spring Boot archetype. There are other Camel archetypes, like `camel-archetype-java` and `camel-archetype-blueprint`. Each archetype creates a boilerplate Camel project on a given runtime.

Looking inside the project

If you look in the new folder you'll see that it has a fairly typical Maven project structure:

```
first-app
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           ├── MySpringBean.java ①
    │   │           ├── MySpringBootApplication.java ②
    │   │           └── MySpringBootRouter.java ③
    │   └── resources
    │       ├── application.properties ④
    │       └── META-INF
    │           ├── LICENSE.txt
    │           └── NOTICE.txt
    └── test
        ├── java
        │   └── com
        │       └── example
        └── resources
```

① A class containing some custom code which we'll call from Camel

② Our application's **"main"** class

③ A `RouteBuilder` class containing our first Camel route

④ A place to store our application's configuration

The RouteBuilder class: where the Camel routes live

The interesting code resides in `MySpringBootRouter.java`. This is a `RouteBuilder` class, which is where you define your Camel routes. The archetype creates a sample route to get you started:

Listing 12. MySpringBootRouter.java - a sample RouteBuilder class

```java
package com.example;

import org.apache.camel.builder.RouteBuilder;
import org.springframework.stereotype.Component;

@Component
public class MySpringBootRouter extends RouteBuilder {

    @Override
    public void configure() {
        // The route is defined here - keep reading to find out more
    }

}
```

When you write Camel routes using the Java DSL, your route definitions go inside a `RouteBuilder` class, by writing a `configure()` method.

If you look at the code, you'll see that the MyRouteBuilder class comes with a route defined inside it already.

The bootstrapping class: where the application starts

For your convenience, the project also includes another Java class, `MySpringBootApplication`, which starts the application:

Listing 13. MySpringBootApplication.java - the bootstrap class

```java
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MySpringBootApplication {

    public static void main(String[] args) {
        SpringApplication.run(MySpringBootApplication.class, args);
    }

}
```

This is a Java `main` class with Spring Boot's bootstrapping features. This allows you to run the application from the command line.

You'll see that we don't explicitly start a Camel Context here. This is because Camel has full support for Spring Boot, and configures this behind-the-scenes for you. All you have to do is define a RouteBuilder class, make it available to Spring Boot (this is usually done with the annotation `@Component`) and your routes will be added into a Camel Context and started.

Now let's look at the code and see what this project does.

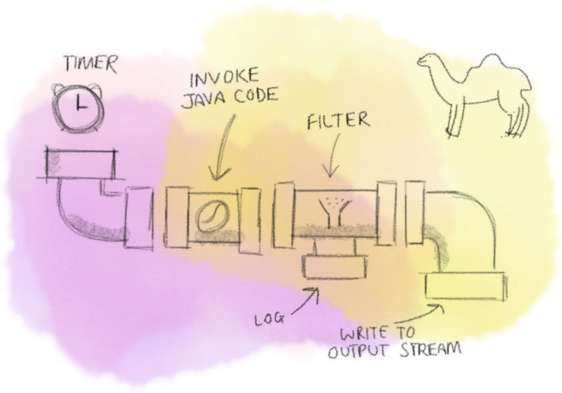
What does the Camel code do?

The demo project defines a Camel route using the code below. As this is a Spring Boot archetype, this example uses the Java DSL:

Listing 14. Demo route created by the archetype

```
from("timer:hello?period={{timer.period}}").routeId("hello")
    .transform().method("myBean", "saySomething")
    .filter(simple("${body} contains 'foo'"))
        .to("log:foo")
    .end()
    .to("stream:out");
```

Let's look at each part of the route in detail:



1. A Timer component (`timer:…`) kicks off the route:

   ```
   from("timer:hello?period={{timer.period}}")
   ```

   Camel's Timer component can be used to fire a route at regular intervals.

   To help you make your routes more configurable, you can use external properties in your Camel routes. Here, the Timer's interval is set to `{{timer.period}}`, which is a reference to a property that is defined in the file `application.properties` as `2000` (2 seconds).

2. The `transform()` EIP states that we want to change the content of the Message.

   ```
   .transform().method("myBean", "saySomething")
   ```

   In this example, we use the `bean()` method, which invokes a method on a Java bean (a Java class).

   ◦ Camel looks for a Java object in the **registry** named `myBean`. Because we're using Spring Boot, it searches the **Spring context** for the bean, which is a registry of the beans that Spring knows about. The bean is in the registry because the `MySpringBean` class has been annotated with `@Component("myBean")`.

   ◦ Camel invokes the method `saySomething()` on that bean.

3. The `filter()` EIP tells Camel to filter the message, based on an expression.

   ```
   .filter(simple("${body} contains 'foo'"))
       .to("log:foo")
   .end()
   ```

   In this case, we use Camel's simple expression language to check whether the message Body contains `foo`. If so, we dump the current exchange to the log.

4. Finally, the content of the message body is written to the output stream (that is, standard out or STDOUT):

   ```
   .to("stream:out");
   ```

Now let's compile the application, run it and see what happens.

Compiling and running

To compile the code, execute this command:

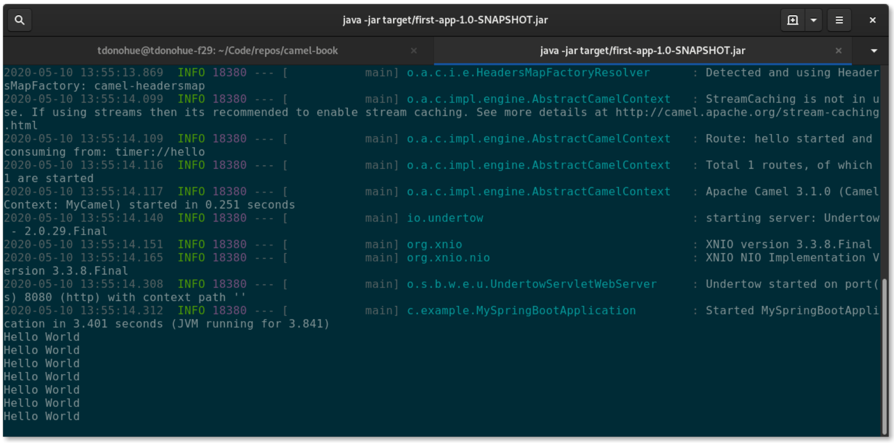
```
mvn clean install
```

This will compile your classes and package them up into a standalone, executable JAR file which is created in the `target/` folder.

Once the application has finished compiling, run it using:

```
java -jar target/first-app-1.0-SNAPSHOT.jar
```

Eventually, once Spring Boot starts up, you will see this in the logs:


java -jar target/first-app-1.0-SNAPSHOT.jar
tdonohue@tdonohue-f29: ~/Code/repos/camel-book
java -jar target/first-app-1.0-SNAPSHOT.jar

```
2020-05-10 13:55:13.869  INFO 18380 --- [           main] o.a.c.i.e.HeadersMapFactoryResolver      : Detected and using Header
sMapFactory: camel-headersmap
2020-05-10 13:55:14.099  INFO 18380 --- [           main] o.a.c.impl.engine.AbstractCamelContext   : StreamCaching is not in u
se. If using streams then its recommended to enable stream caching. See more details at http://camel.apache.org/stream-caching
.html
2020-05-10 13:55:14.109  INFO 18380 --- [           main] o.a.c.impl.engine.AbstractCamelContext   : Route: hello started and 
consuming from: timer://hello
2020-05-10 13:55:14.116  INFO 18380 --- [           main] o.a.c.impl.engine.AbstractCamelContext   : Total 1 routes, of which 
1 are started
2020-05-10 13:55:14.117  INFO 18380 --- [           main] o.a.c.impl.engine.AbstractCamelContext   : Apache Camel 3.1.0 (Camel
Context: MyCamel) started in 0.251 seconds
2020-05-10 13:55:14.140  INFO 18380 --- [           main] io.undertow                              : starting server: Undertow
 - 2.0.29.Final
2020-05-10 13:55:14.151  INFO 18380 --- [           main] org.xnio                                 : XNIO version 3.3.8.Final
2020-05-10 13:55:14.165  INFO 18380 --- [           main] org.xnio.nio                             : XNIO NIO Implementation V
ersion 3.3.8.Final
2020-05-10 13:55:14.308  INFO 18380 --- [           main] o.s.b.w.e.u.UndertowServletWebServer     : Undertow started on port(
s) 8080 (http) with context path ''
2020-05-10 13:55:14.312  INFO 18380 --- [           main] c.example.MySpringBootApplication        : Started MySpringBootAppli
cation in 3.401 seconds (JVM running for 3.841)
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

The logs above tell us that Apache Camel started correctly, read our route definition and the Camel Context is now running.

Every few seconds, you will see the text **Hello World** in the logs.

Camel’s Timer component triggers the route to be executed every few seconds. When the route executes, the message body is set to **Hello World**, and it is written to the console log.

Now that Camel has started, it will continue running until it received an instruction to stop.

To end the app, just press **Ctrl + C**.

🏆 **Level 1 Camel achieved!**

Although this is a very basic demo of Camel, you can begin to see how it works. This demo already shows how Camel integrates with Java code, and how it processes and filters messages. When you start adding components and EIPs, you can build almost any kind of data flow you can think of.

In the next section we’ll see how you can add capabilities to your Camel routes to integrate with other systems, and begin to process your data. Let’s go!

End of Free Edition

This is the end of this free version of the book. I hope you've enjoyed it and you're raring to go with Apache Camel.

When you upgrade to the full version of this book, you'll learn:

- **Camel techniques** - the most common patterns you'll use when developing Camel applications, like transforming messages, scheduling routes, using properties, and calling custom Java code.
- **Developer workflow** - how to develop with Camel, write unit tests and monitor your Camel app.
- How to **integrate** Camel with web services like REST and SOAP
- Common **questions** asked by newbie Camel developers - and full, detailed answers to them!
- A **glossary** of Camel terms, to make things clear

The complete edition of the book also comes with **video lessons and a cheatsheet** to get you started!

To upgrade to the full version of the book now, head on over to:

https://tomd.xyz/camelstepbystep.

If you've found this book useful, please feel free to share it with your friends and coworkers, it helps it to reach more people.

Thanks for reading!

Tom Donohue