# Prerequisites

* **JAVA 8 (newer versions might have issues with examples)**
* **Maven**

# Getting Camel

This part is not mandatory as it’s mainly about getting the whole library, which we’ll be pulling in as a dependency through maven. Camel is available from the official Apache Camel website at <http://camel.apache.org/download.html> . On that page, you’ll see a list of all the Camel releases and the downloads for the latest release.

We’ll be using Camel 2.20.1. To get this version, click the Camel 2.20.1 Release link. Near the bottom of the page, you’ll find two binary distributions: the zip distribution is for Windows users, and the tar.gz distribution is for macOS/Linux users. After you’ve downloaded one of the distributions, extract it to a location on your hard drive. Open a command prompt and go to the location where you extracted the Camel distribution. Issuing a directory listing here will give you something like this:

***tadas@work:~/Documents/Applications/apache-camel-2.23.1$ ls***

***doc examples lib LICENSE.txt NOTICE.txt README.txt***

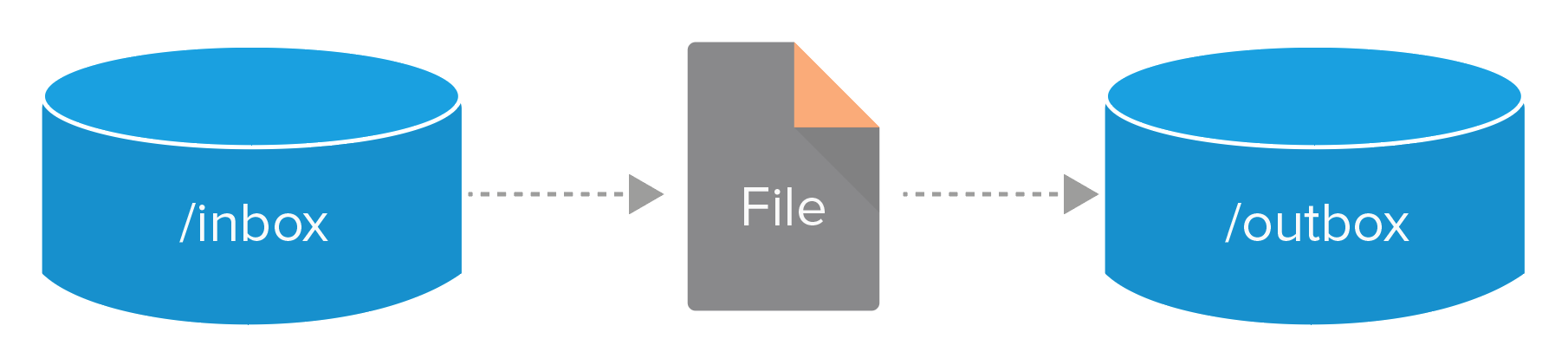
As you can see, the distribution is small, and you can probably guess what each directory contains already. Here are the details:

* **doc** - Contains the Camel manual in HTML format. This manual is a download of a large portion of the Apache Camel website at the time of release. As such, it’s a decent reference for those unable to access the Camel website (or if you misplaced your copy of Camel in Action).
* **examples** - Includes 97 Camel examples.
* **lib** - Contains all Camel libraries.
* **LICENSE.txt** - Contains the license of the Camel distribution. Because this is an Apache project, the license is the Apache License, version 2.0.
* **NOTICE.txt -** Contains copyright information about the third-party dependencies included in the Camel distribution.
* **README.txt** - Contains a short intro to Camel and a list of helpful links to get new users up and running.

# First try at Camel

The first example you’ll look at can be considered the “hello world” of integrations:

routing files. Suppose you need to read files from one directory (/inbox), process them in some way, and write the result to another directory (/outbox). For simplicity, you’ll skip the processing, so your output will be merely a copy of the original file. Picture 2 illustrates this.



Picture 2

In Picture 2 we can see that files are routed from the data/inbox directory to the data/outbox directory. This can be achieved with pure Java without Camel, and it would look like this. You can try these examples by creating a maven project and then creating the two classes, inbox directory, some example files in the inbox directory.

**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 {**

**initiateCopying("inbox", "outbox");**

**}**

**private static void initiateCopying(String inbox, String outbox) throws Exception {**

**File inboxDirectory = new File(inbox); // have this directory in the root of the project**

**File outboxDirectory = new File(outbox); // will be created if the directory doesn't exist**

**outboxDirectory.mkdir(); // creates directory**

**copyFiles(inboxDirectory, outboxDirectory); // initialises file copying**

**}**

**private static void copyFiles(File inbox, File outbox) throws Exception {**

**// goes through all the files in the inbox and copies them to outbox using the copyFile()**

**File[] files = inbox.listFiles();**

**for (File source : files) {**

**if (source.isFile()) {**

**File dest = new File(outbox.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();**

**}**

**}**

**}**

As we can see the Camels way of doing is less complex and shorter. It doesn’t require us to use low level file APIs and ensure that resources are properly closed, a task that can potentially go wrong. You would also need to poll the directory for new files and keep a register so that we don’t copy same files all over again. This would make a simple example more complex. Now let’s take a look at Camels example with polling.

**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 {**

**copyFiles();**

**}**

**private static void copyFiles() throws Exception{**

**CamelContext context = new DefaultCamelContext();**

**context.addRoutes(new RouteBuilder() {**

**public void configure() {**

**from("file:inbox?noop=true").to("file:outbox");**

**}**

**});**

**context.start();**

**Thread.sleep(10000); // need sleep to keep JVM running until the job is done**

**context.stop();**

**}**

**}**

Most of this code is boilerplate. Every Camel application uses a CamelContext that’s subsequently started and stopped. A sleep method is required to allow your simple Camel application time to copy the files. What you should focus on in Camels example is the route.

***from("file:data/inbox?noop=true").to("file:data/outbox");***

Routes in Camel are defined in an easy to understand way when read. This route can be interpreted like this: consume messages from file location /inbox with the noop option

set, and send to file location /outbox. The noop option tells Camel to leave the source file as is. Without this option, the file would be moved. You may also want to note that, excluding the boilerplate code, you created a file-­polling route in just two lines of Java code.

In order to run this example you will need to have a pom.xml file with the following things:

**<?xml version="1.0" encoding="UTF-8"?>**

**<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>**

**<groupId>camel</groupId>**

**<artifactId>qa</artifactId>**

**<version>1.0-SNAPSHOT</version>**

**<properties>**

**<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>**

**<maven.compiler.source>1.7</maven.compiler.source>**

**<maven.compiler.target>1.7</maven.compiler.target>**

**</properties>**

**<dependencies>**

**<dependency>**

**<groupId>org.apache.camel</groupId>**

**<artifactId>camel-core</artifactId>**

**<version>2.22.0</version>**

**</dependency>**

**<dependency>**

**<groupId>org.slf4j</groupId>**

**<artifactId>slf4j-log4j12</artifactId>**

**<version>1.7.25</version>**

**</dependency>**

**</dependencies>**

**</project>**

Also in **src/main/resources** directory a file **log4j.properties** is required with the following settings:

**# uncomment the next line to debug Camel**

**#log4j.logger.org.apache.camel=DEBUG**

**log4j.rootLogger=INFO, out**

**log4j.appender.out=org.apache.log4j.ConsoleAppender**

**log4j.appender.out.layout=org.apache.log4j.PatternLayout**

**log4j.appender.out.layout.ConversionPattern=[%30.30t] %-30.30c{1} %-5p %m%n**

After all of the files have been added, trigger main method of either Camel or Java example in your preferred IDE.

**NOTE:** You may be wondering whether you always need that ugly Thread.sleep call. Thankfully, the answer is no! The example was created in this way to demonstrate the low-level mechanics of Camel’s API. If you were deploying your Camel route to another container or runtime or as a unit test, you wouldn’t need to explicitly wait a set amount of time. Even for standalone routes not deployed to any container, there’s a better way. Camel provides the org.apache.camel.main.Main helper class to start up a route of your choosing and wait for the JVM to terminate.

# Camel’s message model

Camel uses two abstractions for modeling messages:

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

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

# 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 we can see in Picture 3.



Picture 3

Messages have a body (a payload), headers (optional), and optional attachments, as illustrated in picture 4.



Picture 4

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

## Headers and attachments

Headers are values associated with the message, such as sender identifiers, hints about content encoding, authentication information, and so on. Headers are name-value

pairs; the name is a unique, case-insensitive string, and the value is of type java.lang.Object . Camel imposes no constraints on the type of the headers. There are also no

constraints on the size of headers or on the number of headers included with a message. Headers are stored as a map within the message. A message can also have optional

attachments, which are typically used for the web service and email components.

## Body

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

## Fault flag

Messages also have a fault flag. A few protocols and specifications, such as SOAP Web Services, distinguish between output and fault messages. They’re both valid responses

to invoking an operation, but the latter indicates an unsuccessful outcome. In general,

faults aren’t handled by the integration infrastructure. They’re part of the contract between the client and the server and are handled at the application level. During routing, messages are contained in an exchange.

# Exchange

An exchange in Camel is the message’s container during routing. An exchange also

provides support for the various types of interactions between systems, also known as

message exchange patterns (MEPs). MEPs are used to differentiate between one-way and

request-response messaging styles. The Camel exchange holds a pattern property that

can be either of the following:

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



Picture 5

Let’s look at the elements of Picture 5:

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

The exchange is the same for the entire lifecycle of routing, but the messages can

change, for instance, if messages are transformed from one format to another.

# Camel concepts

## Routing Engine

Camel’s routing engine is what moves messages under the hood. This engine isn’t

exposed to the developer, but you should be aware that it’s there and that it does all the

heavy lifting, ensuring that messages are routed properly.

## Routes

Routes are obviously a core abstraction for Camel. The simplest way to define a route is

as a chain of processors. There are many reasons for using routers in messaging applications. By decoupling clients from servers, and producers from consumers, routes can

do the following:

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

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

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

Under the hood, Camel clones the route definition into three separate routes. So, it

behaves similarly to three separate routes as follows:

**from("jms:queue:A").to("jms:queue:D");**

**from("jms:queue:B").to("jms:queue:D");**

**from("jms:queue:C").to("jms:queue:D");**

Even though it’s perfectly legal in Camel 2.x, we don’t recommend using multiple

inputs per route. This ability will be removed in the next major version of Camel. To

define these routes, we use a DSL.

## Domain - specific language

To wire processors and endpoints together to form routes, Camel defines a DSL. The

term DSL is used a bit loosely here. In Camel, DSL means a fluent Java API that contains methods named for EIP terms.

Consider this example:

**from("file:data/inbox")**

**.filter().xpath("/order[not(@test)]")**

**.to("jms:queue:orders");**

Here, in a single Java statement, you define a route that consumes files from a file end-point. Messages are then routed to the filter EIP, which will use an XPath predicate to

test whether the message is not a test order. If a message passes the test, it’s forwarded

to the JMS endpoint. Messages failing the filter test are dropped. Camel provides multiple DSL languages, so you could define the same route by using the XML DSL, like this:

**<route>**

**<from uri="file:data/inbox"/>**

**<filter>**

**<xpath>/order[not(@test)]</xpath>**

**<to uri="jms:queue:orders"/>**

**</filter>**

**</route>**

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

to see what a processor is.

## Processor

The processor is a core Camel concept that represents a node capable of using, creating, or modifying an incoming exchange. During routing, exchanges flow from one processor to another; as such, you can think of a route as a graph having specialized processors as the nodes, and lines that connect the output of one processor to the input of another. Processors could be implementations of EIPs, producers for specific components, or your own custom creation. Picture 6 shows the flow between processors.



Picture 6

A route first starts with a consumer (think “from” in the DSL) that populates the initial

exchange. At each processor step, the out message from the previous step is the in

message of the next. In many cases, processors don’t set an out message, so in this case

the in message is reused. At the end of a route, the MEP of the exchange determines

whether a reply needs to be sent back to the caller of the route. If the MEP is InOnly ,

no reply will be sent back. If it’s InOut, Camel will take the out message from the last

step and return it.

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

## Component

Components are the main extension point in Camel. To date, the Camel ecosystem has more than 280 components that range in function from data transports, to DSLs, to

data formats, and so on. You can even create your own components for Camel.

From a programming point of view, components are fairly simple: they’re associated

with a name that’s used in a URI, and they act as a factory of endpoints. For example,

FileComponent is referred to by file in a URI, and it creates FileEndpoints. The endpoint is perhaps an even more fundamental concept in Camel.

## Endpoint

An endpoint is the Camel abstraction that models the end of a channel through which a system can send or receive messages. This is illustrated in Picture 7.



Picture 7

In Camel, you configure endpoints by using URIs, such as **file:data?delay=5000** , and you also refer to endpoints this way. At runtime, Camel looks up an endpoint based on the URI notation. Picture 8 shows how this works.



Picture 8 (Endpoint URIs are divided into three

parts: a scheme, a context path, and options.)

The scheme (1 part of picture 8), denotes which Camel component handles that type of endpoint. In this case, the scheme of **file** selects **FileComponent** . **FileComponent** then works as a factory, creating **FileEndpoint** based on the remaining parts of the URI. The context path **data** (2 part of picture 8), tells **FileComponent** that the starting folder is data. The option, **delay=5000** (3 part of picture 8) indicates that files should be polled at a 5-second interval.

There’s more to an endpoint than meets the eye. Picture 9 shows how an endpoint

works together with an exchange, producers, and consumers. At first glance, Picture 9 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.



Picture 9

## Consumer

A consumer is the service that receives messages produced by some external system, wraps them in an exchange, and sends them to be processed. Consumers are the

source of the exchanges being routed in Camel. Looking back at Picture 9, you can see where the consumer fits in with other Camel concepts. To create a new exchange, a consumer will use the endpoint that wraps the payload being consumed. A processor is then used to initiate the routing of the exchange in Camel via the routing engine.

Camel has two kinds of consumers: event-driven consumers and polling consumers.

The differences between these consumers are important, because they help solve different problems.

### Event - driven consumer

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

### Polling consumer

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