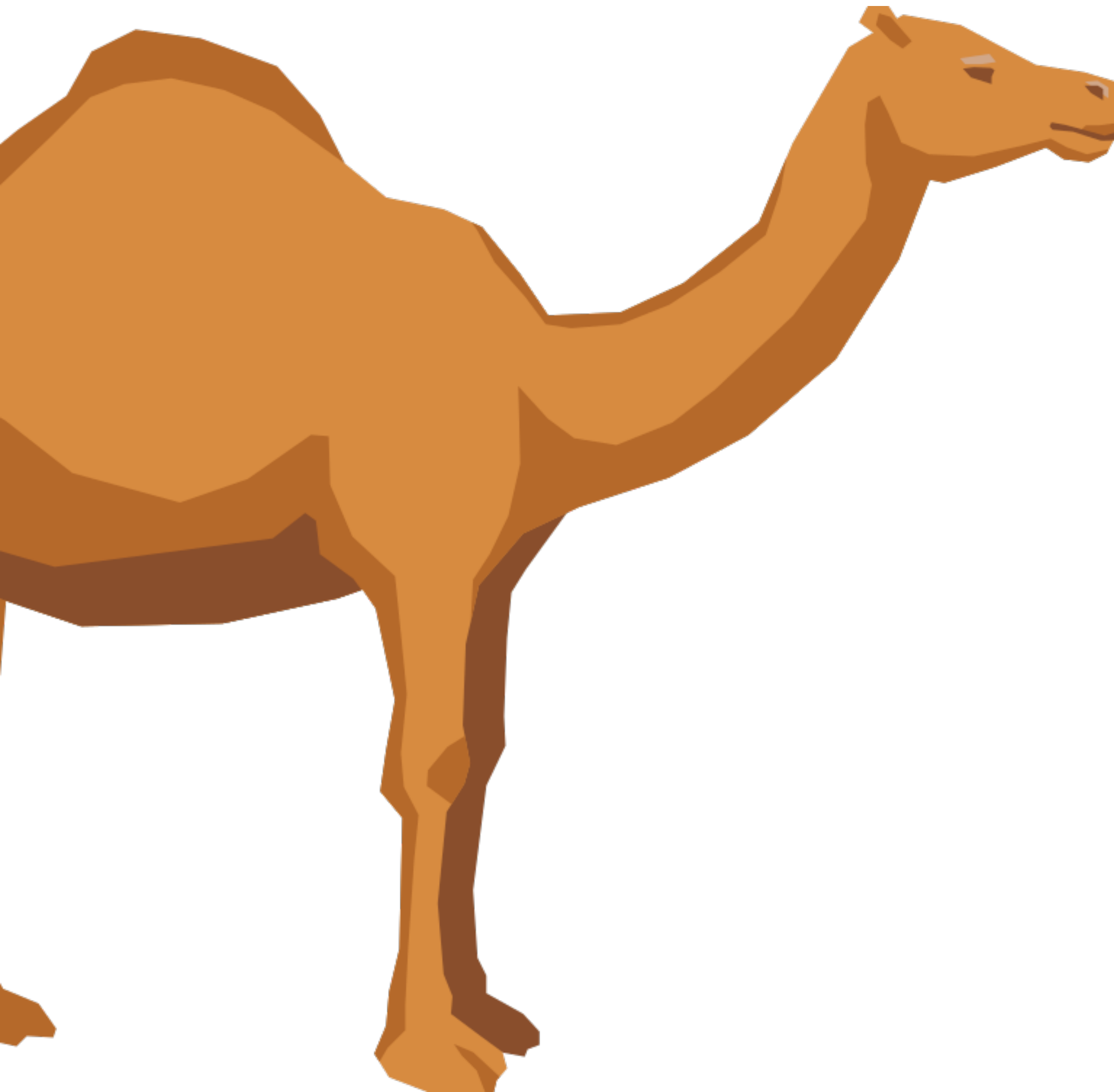


Apache Camel Tutorial

A free ebook from integrationwire.com



1

Introduction

Thanks for downloading this free beginners' Apache Camel tutorial ebook.

Apache Camel is an increasingly popular *integration framework*.

How popular? Well, a question on Stack Overflow, [What exactly is Apache Camel](#), has been viewed over a quarter of a million times.

That's huge! That means not only is there strong interest in Camel, but people are eager to find out what it can do.

You're reading this because you want to get started with Apache Camel and understand the basics. Good choice!

I'm excited to share this tutorial with you. I had fun making it and I hope you find it useful. I created this tutorial with the understanding that not everyone who crosses paths with Apache Camel does so by coming from a hardcore Java development background.

Developers these days come from all sorts of backgrounds. Apache Camel might be one of your first experiences with Java development. Or, you may have come across Apache Camel through one of the commercial offerings that depend on it, such as Talend ESB or Red Hat JBoss Fuse.

Whatever your background, this tutorial will help you get to grips with the very basics of Apache Camel and will get you up and running quickly.

Enjoy!

— Tom Donohue, integrationwire.com

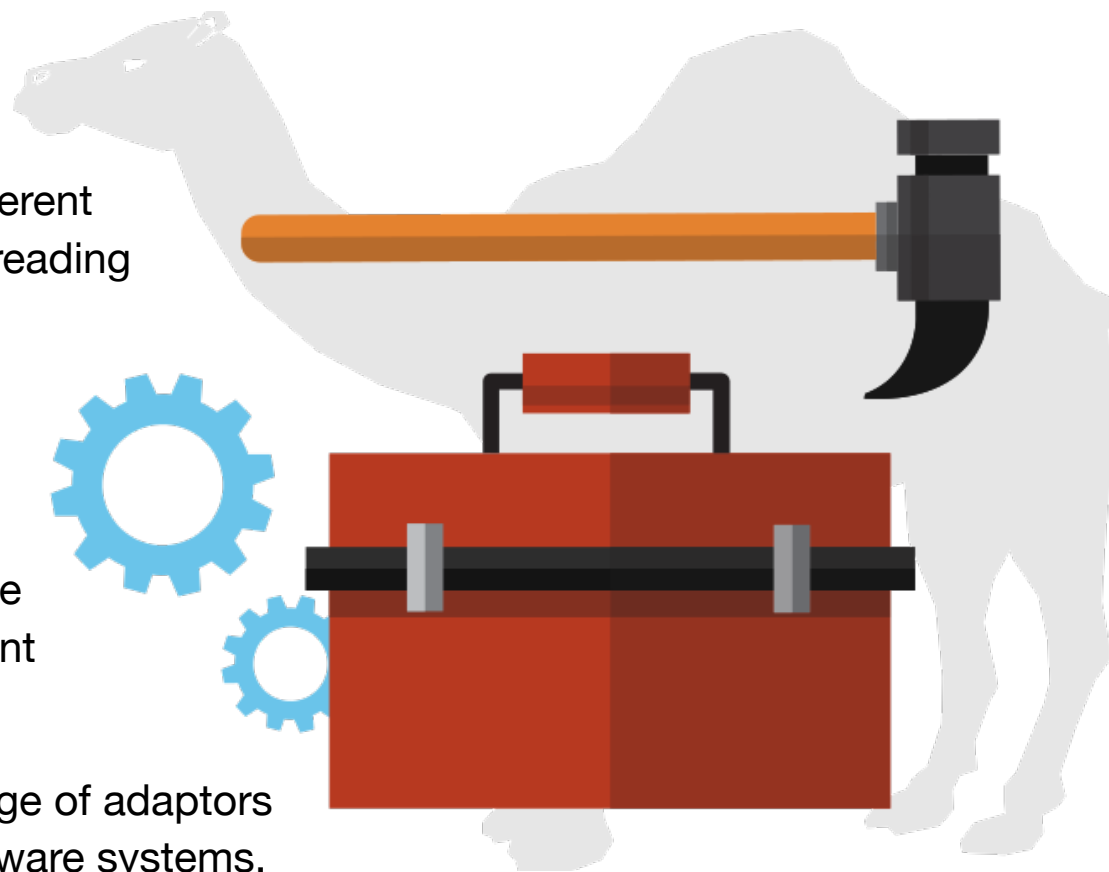
What is Apache Camel?

So what is Apache Camel and what is it used for?

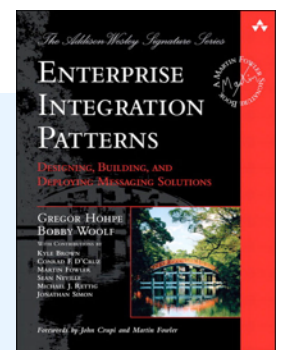
Apache Camel is a framework for Java that you can use to build integrations between different systems. It comes with components that support lots of different types of systems, from web services, to reading and writing files on disk.

You can think of Camel as a **plumbing toolkit** for Java. Just like real plumbing pipes, Camel takes data from one point, and pipes 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 that fit all sorts of different apps and software systems. With a full range of tools at your disposal, it's then up to you how you choose to build the plumbing.



Camel was largely inspired by the book Enterprise Integration Patterns, a sort of academic textbook for integrating software systems. The authors of the book (now considered a 'classic' in its sphere) took dozens of common use cases and distilled them into reusable patterns, describing their use and, in some cases, providing suggested code for how to implement them.



The Camel developers thought it would be a great idea to build a Java framework that represented the ideals of the book. And so Apache Camel borrows heavily from the book.

Some things worth noting about Camel:

- It's **built in Java** – this might seem an obvious point to make, but once you understand this you'll see that you have the full power of Java at your disposal
- The **source code is completely open** – check it out at Github
- It's **not just for web services** – it's a general integration framework. This means you might choose to use Camel to interact with web services, but you don't have to.
- It comes with a **huge library of components** – if you can think of a system you'd like to interact with, somebody has probably already written a component for it; everything from pushing files to an Amazon cloud, to sending a tweet.
- It's **mature** – Apache Camel has truly come of age and forms the foundation of some commercially-sold integration products, like Red Hat Fuse ESB and Talend ESB.

What is Apache Camel used for?

Almost any time you need to move data from A to B, you can probably use Camel. Think of the following examples:

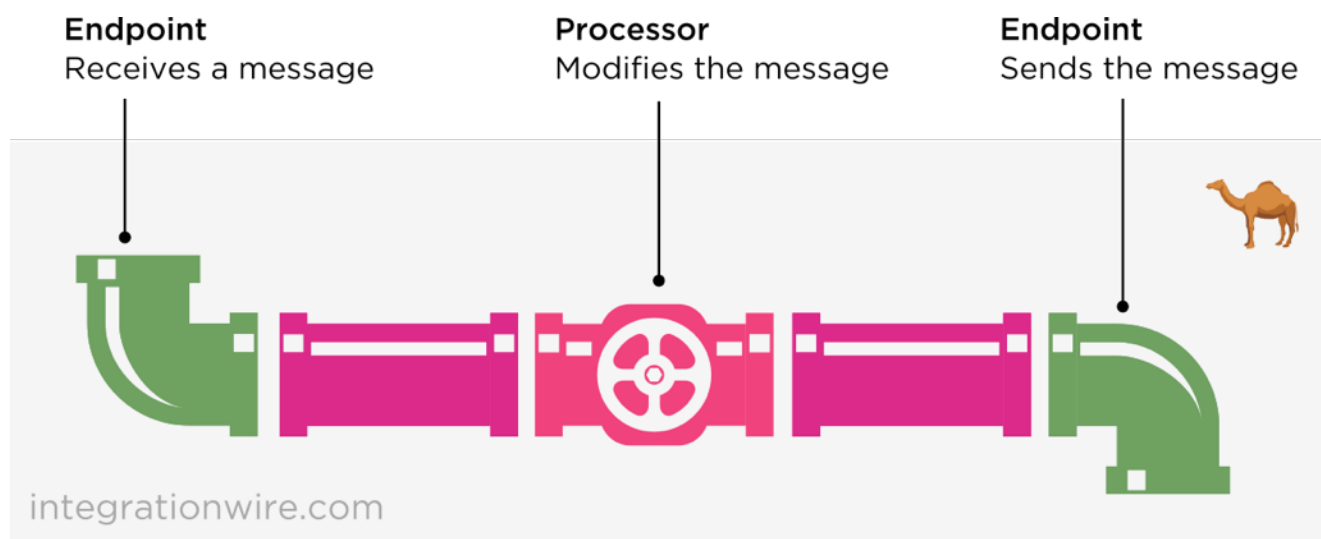
- Picking up invoices from an FTP server and emailing them to you
- Taking files from a folder and pushing them into Google Drive
- Taking messages from a JMS queue and using them to invoke a web service
- Making a web service that allows users to retrieve customer details from a database

These are just a few examples. With the wide range of components available in Camel, the sky's the limit.

Thinking in Camel

The basic concept in Camel is the **route**.

Thinking in terms of our plumbing analogy, a **route** is like a **pipe** that moves data from one place to another. These places are called **endpoints**.



To allow Camel to connect to an endpoint, it comes with a library of **components**. A component is simply a type of plug that allows you to connect to an endpoint (such as a system, application, etc).

So Camel allows you to create routes that move data between endpoints, using components. This is done through Java code, more of which you'll see later on.

Components are like off-the-shelf plugs that you can use in your routes. Any time you need to move data to or from an application, you'll probably find that a component has already been written 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.

The beauty is that these components are reusable, and you can even contribute your own.

In between endpoints, the data can also be transformed or modified, either by passing the data through another **endpoint**, or by using a **processor**.

Working with messages in Camel

Camel works with data **using a message model**.

What does this mean? It means that Camel doesn't treat data like a stream, like water flowing through a pipe.

Instead, it receives data as individual messages – like packages flowing through a post office.

Each message is treated as an individual unit. A message could be huge, or it could be very small. Each message is wrapped into a standard Camel object called a **Message**. A **Message** has **Headers**, which can be used to hold values associated with the message, and a **Body**, where the main message data itself lies.

That **Message** object is then passed along the route.

The important thing to understand in Camel's message model is that the Body can contain almost any kind of Java object – it doesn't have to be XML, or even plain text for that matter. You can just as easily route a binary file, or even something more complicated.

2

The Tutorial

You will need:

- Eclipse, installed on your machine
- Apache Maven
- that's it!

Note for Mac users: If you're using a Mac, I recommend using the Homebrew package manager to help you install Maven. Follow the instructions at <http://brew.sh/> to install. Then, once Homebrew is installed, install Maven by typing `brew install maven` from a Terminal window.

Once Maven is installed, you can run `mvn -version` in a Terminal window to check it's been installed correctly.

Creating the Camel project

First you'll create the Maven project that will hold your Apache Camel code. Maven is a build tool for Java based projects. It can be used to create new projects, automatically manage dependencies, build, compile and deploy code, and much more. Many projects adopt Maven as a standard.

Maven is immensely useful when working with Camel because it can create a new Camel project from a template (known as an "archetype"), with some boilerplate code to get you up and running. Maven can also turn this code into a project that can be managed in Eclipse.

Creating the Camel maven project

The first step is to create the new project, from the Maven command line. Follow these steps to create your example Camel Maven project:

1. Ensure Maven is installed and on your path.
2. Open a Command Prompt (Windows), or Terminal (Mac)
3. Enter the following command:

```
mvn archetype:generate  
-DarchetypeGroupId=org.apache.camel.archetypes  
-DarchetypeArtifactId=camel-archetype-java
```

This tells Maven to generate a project using the archetype camel-archetype-java, which creates a basic Camel Java project

4. When prompted, enter the following:
 1. groupId: `org.example`
 2. artifactId: `MyCamelProject`
 3. version: **(press Enter to accept 1.0-SNAPSHOT)**
 4. package: **(press Enter)**

5. When you're returned to the command line, type `cd MyCamelProject`
6. Type the following to convert the project into an Eclipse project: `mvn eclipse:eclipse`

Here's the output from Maven (it's very verbose!):

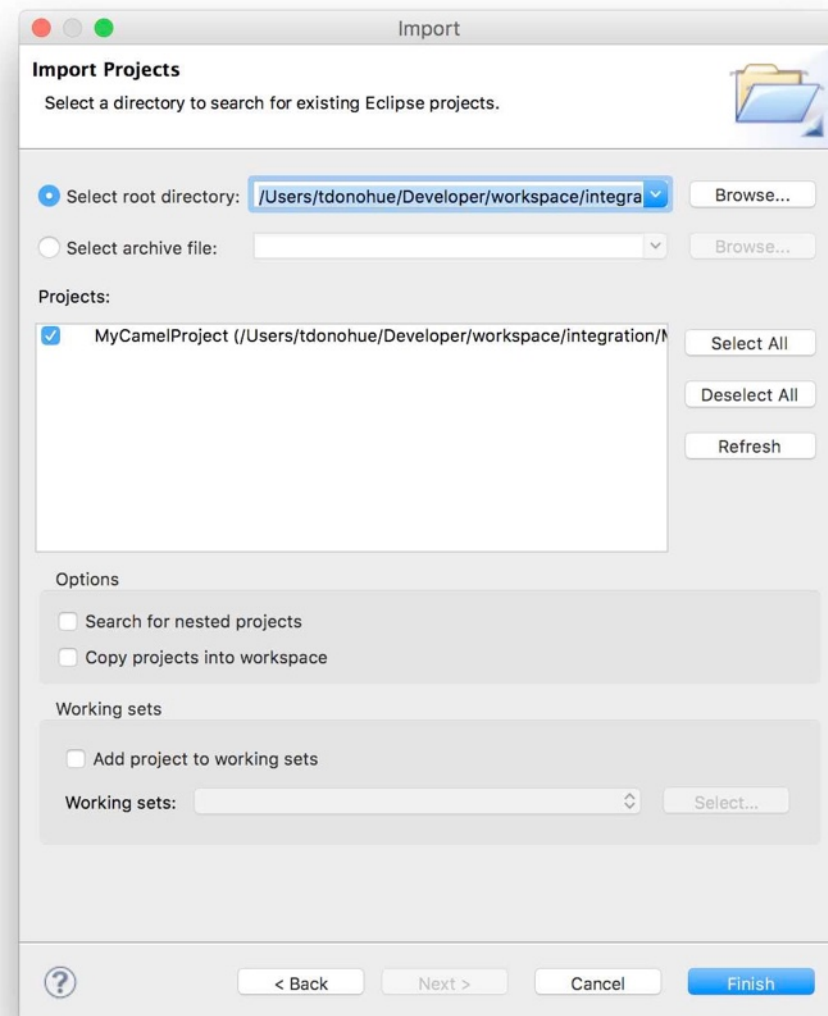
```
bubbs:integration tdonohue$ mvn archetype:generate -DarchetypeGroupId=org.apache.camel.archetypes
-DarchetypeArtifactId=camel-archetype-java
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[INFO]
[INFO] >>> maven-archetype-plugin:2.4:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.4:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.4:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype [org.apache.camel.archetypes:camel-archetype-java:2.16.1] found in catalog remote
Define value for property 'groupId': : org.example
Define value for property 'artifactId': : MyCamelProject
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': org.example: :
[INFO] Using property: camel-version = 2.16.1
[INFO] Using property: exec-maven-plugin-version = 1.2.1
[INFO] Using property: log4j-version = 1.2.17[INFO] Using property: maven-compiler-plugin-version = 3.3
[INFO] Using property: maven-resources-plugin-version = 2.6
[INFO] Using property: slf4j-version = 1.7.12
Confirm properties configuration:
groupId: org.example
artifactId: MyCamelProject
version: 1.0-SNAPSHOT
package: org.example
camel-version: 2.16.1
exec-maven-plugin-version: 1.2.1
log4j-version: 1.2.17
maven-compiler-plugin-version: 3.3c
maven-resources-plugin-version: 2.6
slf4j-version: 1.7.12
```

Importing the project into Eclipse

Now we can import the project into Eclipse:

1. Open Eclipse
2. Choose File → Import
3. On the Import dialog box, select General → Existing Projects into Workspace. Click Next.
4. Use the Browse button by Select root directory to find your project root directory. Select your project and then click Finish to import.

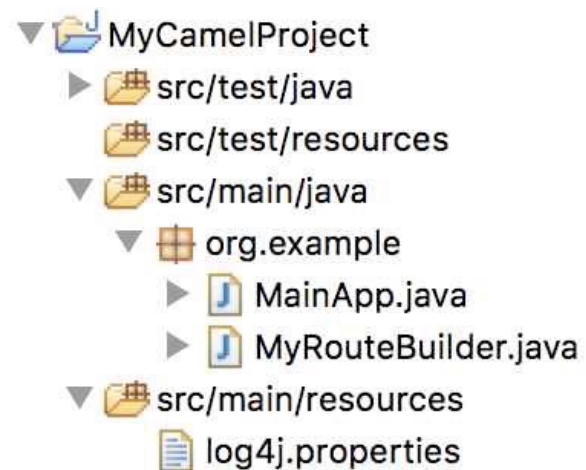
Now your Camel project is in Eclipse. Shortly we'll run the route. But first let's take a look at the structure of a typical Camel project.



Typical Camel project structure

A Camel project created from an archetype will look like this.

The main code resides in `MyRouteBuilder.java`. In Camel-speak, this is a `RouteBuilder` class, and this is where your plumbing configuration resides.



A `RouteBuilder` is where your plumbing instructions go. A `RouteBuilder` class contains route definitions, which describe each pipe that you want Camel to build.

Each pipe, or route, starts with `from(url)` where url expresses the endpoint where the data is coming from. A route may consist of multiple steps, such as transforming the data or logging it. But a route will usually end with a `to()` instruction, which expresses where the data will be delivered to.

A really simple route could look something like this:

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

This would move all incoming files in the customers/new folder, to the customers/old folder.

You'll see that the included `MyRouteBuilder` comes with a route already defined. This means that a route has already been configured.

A route is “always-on” – that is to say, once the route is started, it continues to listen for and process messages, **until it is shut down**.

For convenience, the Maven archetype also includes another Java class, `MainApp`, which can be used to run your route from with Eclipse. It uses the standard Java `main()` method to initialise `MyRouteBuilder` and create the routes.

This is useful for testing, as you'll see next.

Running the demo route

Understanding the demo project

The demo project includes some code that builds a simple Camel route. This section describes what the demo route does.

The demo project defines a route using the code below. The code is given in a special Camel syntax, or DSL (Domain-specific Language):

```
public void configure() {  
  
    // here is a sample which processes the input files  
    // (leaving them in place - see the 'noop' flag)  
    // then performs content based routing on the message using XPath  
    from("file:src/data?noop=true")  
        .choice()  
            .when(xpath("/person/city = 'London'"))  
                .log("UK message")  
                .to("file:target/messages/uk")  
            .otherwise()  
                .log("Other message")  
                .to("file:target/messages/others");  
}
```

This code defines a route that picks up files from the `src/data` folder, checks to see whether the message XML element `<person><city>...</city></person>` has the value London, and then routes the message to a `target/messages/uk` folder if so, or otherwise to the folder `target/messages/others`.

Let's look at each part of this statement in detail:

- The `from()` method starts the route. The part in brackets defines the URI of the endpoint. In this case, the endpoint is a `file:` component, and the following part defines the folder to consume from, `src/data`.

- The `choice()` method states that the message is about to be routed, according to a condition.
- The `when()` method checks whether the city element in the XML message equals the string `"London"`. If so, it writes a log message, `"UK message"`, and then moves the file to the `target/messages/uk` folder.
- The `otherwise()` method defines what should happen if the previous test fails. If so, it writes a log message, `"Other message"`, and moves the file to the `target/messages/others` folder.

You'll see that although it's been formatted for easier reading, the entire route is a single Java statement – it starts with the method call `from()`, and ends with a `to()` method call and the statement terminator, `;`.

To get you started, the demo route provides a couple of demo files in the folder `src/data`. The two files are `message1.xml` and `message2.xml`.

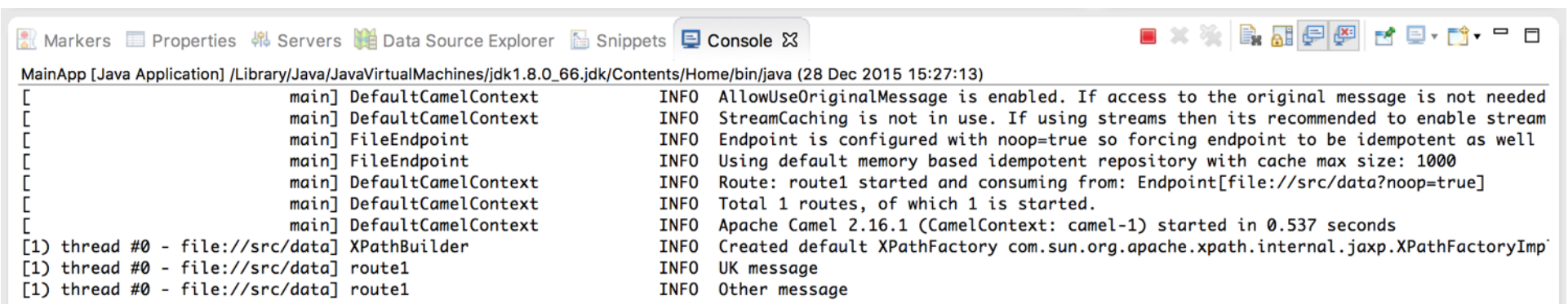
When the route runs, these two files will be picked up by Camel, inspected to see whether they are London files, and routed accordingly.

Now let's run the route and see what happens.

Running the route

To run the route:

1. In the project tree, find `org.example` → `MainApp.java`.
2. Right-click the file and select `Run As` → `Java Application`.
3. If you're prompted by a `Select Java Application` dialog box, tell Eclipse to run the app as a Camel route by selecting `Main – org.apache.camel` from the list.
4. Camel will launch, process the files, and you'll see the output from the route in the `Console` window.



```

MainApp [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/bin/java (28 Dec 2015 15:27:13)
[ main] DefaultCamelContext INFO AllowUseOriginalMessage is enabled. If access to the original message is not needed
[ main] DefaultCamelContext INFO StreamCaching is not in use. If using streams then its recommended to enable stream
[ main] FileEndpoint INFO Endpoint is configured with noop=true so forcing endpoint to be idempotent as well
[ main] FileEndpoint INFO Using default memory based idempotent repository with cache max size: 1000
[ main] DefaultCamelContext INFO Route: route1 started and consuming from: Endpoint[file://src/data?noop=true]
[ main] DefaultCamelContext INFO Total 1 routes, of which 1 is started.
[ main] DefaultCamelContext INFO Apache Camel 2.16.1 (CamelContext: camel-1) started in 0.537 seconds
[1] thread #0 - file://src/data] XPathBuilder INFO Created default XPathFactory com.sun.org.apache.xpath.internal.jaxp.XPathFactoryImp
[1] thread #0 - file://src/data] route1 INFO UK message
[1] thread #0 - file://src/data] route1 INFO Other message
  
```

So what just happened?

The first file, `message1.xml`, had a `city` value of *London*, so it was routed to the `uk` folder and a log message of *UK message* was written to the console.

The second file, `message2.xml`, had a `city` value of *Tampa*, so it was moved to the `others` folder, and a log message of *Other message* was written.