Apache Camel

USER GUIDE

Version 2.23.0

Table of Contents

CHAPTER 1

Introduction

Apache Camel ™ is a versatile open-source integration framework based on known Enterprise Integration Patterns.
Camel empowers you to define routing and mediation rules in a variety of domain-specific languages, including a Java-based Fluent API, Spring or Blueprint XML Configuration files, and a Scala DSL. This means you get smart completion of routing rules in your IDE, whether in a Java, Scala or XML editor.

Apache Camel uses URIs to work directly with any kind of Transport or messaging model such as HTTP, ActiveMQ, JMS, JBI, SCA, MINA or CXF, as well as pluggable Components and Data Format options. Apache Camel is a small library with minimal dependencies for easy embedding in any Java application. Apache Camel lets you work with the same API regardless which kind of Transport is used - so learn the API once and you can interact with all the Components provided out-of-box.

Apache Camel provides support for Bean Binding and seamless integration with popular frameworks such as CDI, Spring, Blueprint and Guice. Camel also has extensive support for unit testing your routes.

The following projects can leverage Apache Camel as a routing and mediation engine:

- Apache ServiceMix - a popular distributed open source ESB and JBI container
- Apache ActiveMQ - a mature, widely used open source message broker
- Apache CXF - a smart web services suite (JAX-WS and JAX-RS)
- Apache Karaf - a small OSGi based runtime in which applications can be deployed
- Apache MINA - a high-performance NIO-driven networking framework

So don't get the hump - try Camel today! 🙂
Too many buzzwords - what exactly is Camel?
Okay, so the description above is technology focused.
There's a great discussion about Camel at Stack Overflow. We suggest you view the post, read the comments, and browse the suggested links for more details.

CHAPTER 2

Quickstart

To start using Apache Camel quickly, you can read through some simple examples in this chapter. For readers who would like a more thorough introduction, please skip ahead to Chapter 3.

WALK THROUGH AN EXAMPLE CODE

This mini-guide takes you through the source code of a simple example.

Camel can be configured either by using Spring or directly in Java - which this example does.

This example is available in the `examples\camel-example-jms-file` directory of the Camel distribution.

We start with creating a CamelContext - which is a container for Components, Routes etc: {snippet:id=e1|lang=java|url=camel/trunk/examples/camel-example-jms-file/src/main/java/org/apache/camel/example/jmstofile/CamelJmsToFileExample.java}There is more than one way of adding a Component to the CamelContext. You can add components implicitly - when we set up the routing - as we do here for the FileComponent:{snippet:id=e3|lang=java|url=camel/trunk/examples/camel-example-jms-file/src/main/java/org/apache/camel/example/jmstofile/CamelJmsToFileExample.java}or explicitly - as we do here when we add the JMS Component:{snippet:id=e2|lang=java|url=camel/trunk/examples/camel-example-jms-file/src/main/java/org/apache/camel/example/jmstofile/CamelJmsToFileExample.java}The above works with any JMS provider. If we know we are using ActiveMQ we can use an even simpler form using the `activeMQComponent()` method while specifying the brokerURL used to connect to ActiveMQ

In normal use, an external system would be firing messages or events directly into Camel through one if its Components but we are going to use the ProducerTemplate which is a really easy way for testing your configuration:{snippet:id=e4|lang=java|url=camel/trunk/examples/camel-example-jms-file/src/main/java/org/apache/camel/example/jmstofile/CamelJmsToFileExample.java}Next you **must** start the camel context. If you are using Spring to configure the camel context this is automatically done for you; though if you are using a pure Java approach then you just need to call the start() method
camelContext.start();
This will start all of the configured routing rules.

So after starting the CamelContext, we can fire some objects into camel: {snippet:id=e5|lang=java|url=camel/trunk/examples/camel-example-jms-file/src/main/java/org/apache/camel/example/jmstofile/CamelJmsToFileExample.java}

WHAT HAPPENS?

From the ProducerTemplate - we send objects (in this case text) into the CamelContext to the Component *test-jms:queue:test.queue*. These text objects will be converted automatically into JMS Messages and posted to a JMS Queue named *test.queue*. When we set up the Route, we configured the FileComponent to listen off the *test.queue*.

The File FileComponent will take messages off the Queue, and save them to a directory named *test*. Every message will be saved in a file that corresponds to its destination and message id.

Finally, we configured our own listener in the Route - to take notifications from the FileComponent and print them out as text.

That's it!

If you have the time then use 5 more minutes to Walk through another example that demonstrates the Spring DSL (XML based) routing.

WALK THROUGH ANOTHER EXAMPLE

Introduction

Continuing the walk from our first example, we take a closer look at the routing and explain a few pointers - so you won't walk into a bear trap, but can enjoy an after-hours walk to the local pub for a large beer 😉

First we take a moment to look at the Enterprise Integration Patterns - the base pattern catalog for integration scenarios. In particular we focus on Pipes and filters - a central pattern. This is used to route messages through a sequence of processing steps, each performing a specific function - much like the Java Servlet Filters.

Pipes and filters

In this sample we want to process a message in a sequence of steps where each steps can perform their specific function. In our example we have a JMS queue for receiving new orders. When an order is received we need to process it in several steps:

- validate
- register
- send confirm email

This can be created in a route like this:

```
<route>
   <from uri="jms:queue:order"/>
   <pipeline>
      <bean ref="validateOrder"/>
      <bean ref="registerOrder"/>
      <bean ref="sendConfirmEmail"/>
   </pipeline>
</route>
```

Pipeline is default
In the route above we specify `pipeline` but it can be omitted as its default, so you can write the route as:

```
<route>
   <from uri="jms:queue:order"/>
   <bean ref="validateOrder"/>
   <bean ref="registerOrder"/>
   <bean ref="sendConfirmEmail"/>
</route>
```

This is commonly used not to state the pipeline.

An example where the pipeline needs to be used, is when using a multicast and "one" of the endpoints to send to (as a logical group) is a pipeline of other endpoints. For example.

```
<route>
   <from uri="jms:queue:order"/>
   <multicast>
     <to uri="log:org.company.log.Category"/>
     <pipeline>
       <bean ref="validateOrder"/>
       <bean ref="registerOrder"/>
       <bean ref="sendConfirmEmail"/>
     </pipeline>
   </multicast>
</route>
```

The above sends the order (from `jms:queue:order`) to two locations at the same time, our log component, and to the "pipeline" of beans which goes one to the other. If you consider the opposite, sans the `<pipeline>`

```
<route>
   <from uri="jms:queue:order"/>
   <multicast>
     <to uri="log:org.company.log.Category"/>
     <bean ref="validateOrder"/>
     <bean ref="registerOrder"/>
     <bean ref="sendConfirmEmail"/>
   </multicast>
</route>
```

you would see that multicast would not "flow" the message from one bean to the next, but rather send the order to all 4 endpoints (1x log, 3x bean) in parallel, which is not (for this example) what we want. We need the message to flow to the validateOrder, then to the registerOrder, then the sendConfirmEmail so adding the pipeline, provides this facility.
Where as the `bean ref` is a reference for a spring bean id, so we define our beans using regular Spring XML as:

```
   <bean id="validateOrder" class="com.mycompany.MyOrderValidator"/>
```

Our validator bean is a plain POJO that has no dependencies to Camel what so ever. So you can implement this POJO as you like. Camel uses rather intelligent Bean Binding to invoke your POJO with the payload of the received message. In this example we will **not** dig into this how this happens. You should return to this topic later when you got some hands on experience with Camel how it can easily bind routing using your existing POJO beans.

So what happens in the route above. Well when an order is received from the JMS queue the message is routed like Pipes and filters:
1. payload from the JMS is sent as input to the validateOrder bean
2. the output from validateOrder bean is sent as input to the registerOrder bean
3. the output from registerOrder bean is sent as input to the sendConfirmEmail bean

Using Camel Components

In the route lets imagine that the registration of the order has to be done by sending data to a TCP socket that could be a big mainframe. As Camel has many Components we will use the camel-mina component that supports TCP connectivity. So we change the route to:

```
<route>
   <from uri="jms:queue:order"/>
   <bean ref="validateOrder"/>
   <to uri="mina:tcp://mainframeip:4444?textline=true"/>
   <bean ref="sendConfirmEmail"/>
</route>
```

What we now have in the route is a `to` type that can be used as a direct replacement for the bean type. The steps is now:
1. payload from the JMS is sent as input to the validateOrder bean
2. the output from validateOrder bean is sent as text to the mainframe using TCP
3. the output from mainframe is sent back as input to the sendConfirmEmai bean

What to notice here is that the `to` is not the end of the route (the world 😉) in this example it's used in the middle of the Pipes and filters. In fact we can change the `bean` types to `to` as well:

```
<route>
   <from uri="jms:queue:order"/>
   <to uri="bean:validateOrder"/>
   <to uri="mina:tcp://mainframeip:4444?textline=true"/>
   <to uri="bean:sendConfirmEmail"/>
</route>
```

As the `to` is a generic type we must state in the uri scheme which component it is. So we must write **bean:** for the Bean component that we are using.

Conclusion

This example was provided to demonstrate the Spring DSL (XML based) as opposed to the pure Java DSL from the first example. And as well to point about that the `to` doesn't have to be the last node in a route graph.

This example is also based on the **in-only** message exchange pattern. What you must understand as well is the **in-out** message exchange pattern, where the caller expects a response. We will look into this in another example.

See also

- Examples
- Tutorials
- User Guide

CHAPTER 3

Getting Started with Apache Camel

THE *ENTERPRISE INTEGRATION PATTERNS* (EIP) BOOK

The purpose of a "patterns" book is not to advocate new techniques that the authors have invented, but rather to document existing best practices within a particular field. By doing this, the authors of a patterns book hope to spread knowledge of best practices and promote a vocabulary for discussing architectural designs.
One of the most famous patterns books is *Design Patterns: Elements of Reusable Object-oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, commonly known as the "Gang of Four" (GoF) book. Since the publication of *Design Patterns*, many other pattern books, of varying quality, have been written. One famous patterns book is called *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* by Gregor Hohpe and Bobby Woolf. It is common for people to refer to this book by its initials *EIP*. As the subtitle of EIP suggests, the book focuses on design patterns for asynchronous messaging systems. The book discusses 65 patterns. Each pattern is given a textual name and most are also given a graphical symbol, intended to be used in architectural diagrams.

THE CAMEL PROJECT

Camel (http://camel.apache.org) is an open-source, Java-based project that helps the user implement many of the design patterns in the EIP book. Because Camel implements many of the design patterns in the EIP book, it would be a good idea for people who work with Camel to have the EIP book as a reference.

ONLINE DOCUMENTATION FOR CAMEL

The documentation is all under the Documentation category on the right-side menu of the Camel website (also available in PDF form. Camel-related books are also available, in particular the Camel in Action book, presently serving as the Camel bible--it has a free Chapter One (pdf), which is highly recommended to read to get more familiar with Camel.

A useful tip for navigating the online documentation

The breadcrumbs at the top of the online Camel documentation can help you navigate between parent and child subsections.
For example, If you are on the "Languages" documentation page then the left-hand side of the reddish bar contains the following links.

```
Apache Camel > Documentation > Architecture > Languages
```

As you might expect, clicking on "Apache Camel" takes you back to the home page of the Apache Camel project, and clicking on "Documentation" takes you to the main documentation page. You can interpret the "Architecture" and "Languages" buttons as indicating you are in the "Languages" section of the "Architecture" chapter. Adding browser bookmarks to pages that you frequently reference can also save time.

ONLINE JAVADOC DOCUMENTATION

The Apache Camel website provides Javadoc documentation. It is important to note that the Javadoc documentation is spread over several *independent* Javadoc hierarchies rather than being all contained in a single Javadoc hierarchy. In particular, there is one Javadoc hierarchy for the *core* APIs of Camel, and a separate Javadoc hierarchy for each component technology supported by Camel. For example, if you will be using Camel with ActiveMQ and FTP then you need to look at the Javadoc hierarchies for the core API and Spring API.

CONCEPTS AND TERMINOLOGY FUNDAMENTAL TO CAMEL

In this section some of the concepts and terminology that are fundamental to Camel are explained. This section is not meant as a complete Camel tutorial, but as a first step in that direction.

Endpoint

The term *endpoint* is often used when talking about inter-process communication. For example, in client-server communication, the client is one endpoint and the server is the other endpoint. Depending on the context, an endpoint might refer to an *address*, such as a host:port pair for TCP-based communication, or it might refer to a *software entity* that is contactable at that address. For example, if somebody uses "www.example.com:80" as an example of an endpoint, they might be referring to the actual port at that host name (that is, an address), or they might be referring to the web server (that is, software contactable at that address). Often, the distinction between the address and software contactable at that address is not an important one.
Some middleware technologies make it possible for several software entities to be contactable at the same physical address. For example, CORBA is an object-oriented, remote-procedure-call (RPC) middleware standard. If a CORBA server process contains several objects then a client can communicate with any of these objects at the same *physical* address (host:port), but a client communicates with a particular object via that object's *logical* address (called an *IOR* in CORBA terminology), which consists of the physical address (host:port) plus an id that uniquely identifies the object within its server process. (An IOR contains some additional information that is not relevant to this present discussion.) When talking about CORBA, some people may use the term "endpoint" to refer to a CORBA server's *physical address*, while other people may use the term to refer to the *logical address* of a single CORBA object, and other people still might use the term to refer to any of the following:

- The physical address (host:port) of the CORBA server process
- The logical address (host:port plus id) of a CORBA object.
- The CORBA server process (a relatively heavyweight software entity)
- A CORBA object (a lightweight software entity)

Because of this, you can see that the term *endpoint* is ambiguous in at least two ways. First, it is ambiguous because it might refer to an address or to a software entity contactable at that address. Second, it is ambiguous in the *granularity* of what it refers to: a heavyweight versus lightweight software entity, or physical address versus logical address. It is useful to understand that different people use the term *endpoint* in slightly different (and hence ambiguous) ways because Camel's usage of this term might be different to whatever meaning you had previously associated with the term.
Camel provides out-of-the-box support for endpoints implemented with many different communication technologies. Here are some examples of the Camel-supported endpoint technologies.

- A JMS queue.
- A web service.
- A file. A file may sound like an unlikely type of endpoint, until you realize that in some systems one application might write information to a file and, later, another application might read that file.
- An FTP server.
- An email address. A client can send a message to an email address, and a server can read an incoming message from a mail server.
- A POJO (plain old Java object).

In a Camel-based application, you create (Camel wrappers around) some endpoints and connect these endpoints with *routes*, which I will discuss later in Section 4.8 ("Routes, RouteBuilders and Java DSL"). Camel defines a Java interface called `Endpoint`. Each Camel-supported endpoint has a class that implements this `Endpoint` interface. As I discussed in Section 3.3 ("Online Javadoc documentation"), Camel provides a separate Javadoc hierarchy for each communications technology supported by Camel. Because of this, you will find documentation on, say, the `JmsEndpoint` class in the JMS Javadoc hierarchy, while documentation for, say, the `FtpEndpoint` class is in the FTP Javadoc hierarchy.

CamelContext

A `CamelContext` object represents the Camel runtime system. You typically have one `CamelContext` object in an application. A typical application executes the following steps.

1. Create a `CamelContext` object.
2. Add endpoints – and possibly Components, which are discussed in Section 4.5 ("Components") – to the `CamelContext` object.
3. Add routes to the `CamelContext` object to connect the endpoints.
4. Invoke the `start()` operation on the `CamelContext` object. This starts Camel-internal threads that are used to process the sending, receiving and processing of messages in the endpoints.
5. Eventually invoke the `stop()` operation on the `CamelContext` object. Doing this gracefully stops all the endpoints and Camel-internal threads.

Note that the `CamelContext.start()` operation does not block indefinitely. Rather, it starts threads internal to each `Component` and `Endpoint` and then `start()` returns. Conversely, `CamelContext.stop()` waits for all the threads internal to each `Endpoint` and `Component` to terminate and then `stop()` returns.
If you neglect to call `CamelContext.start()` in your application then messages will not be processed because internal threads will not have been created.
If you neglect to call `CamelContext.stop()` before terminating your application then the application may terminate in an inconsistent state. If you neglect to call `CamelContext.stop()` in a JUnit test then the test may fail due to messages not having had a chance to be fully processed.

CamelTemplate

Camel used to have a class called `CamelClient`, but this was renamed to be `CamelTemplate` to be similar to a naming convention used in some other open-source projects, such as the `TransactionTemplate` and `JmsTemplate` classes in Spring.
The `CamelTemplate` class is a thin wrapper around the `CamelContext` class. It has methods that send a `Message` or `Exchange` – both discussed in Section 4.6 ("Message and Exchange")) – to an `Endpoint` – discussed in Section 4.1 ("Endpoint"). This provides a way to enter messages into source endpoints, so that the messages will move along routes – discussed in Section 4.8 ("Routes, RouteBuilders and Java DSL") – to destination endpoints.

The Meaning of URL, URI, URN and IRI

Some Camel methods take a parameter that is a *URI* string. Many people know that a URI is "something like a URL" but do not properly understand the relationship between URI and URL, or indeed its relationship with other acronyms such as IRI and URN.
Most people are familiar with *URLs* (uniform resource locators), such as "http://...", "ftp://...", "mailto:...". Put simply, a URL specifies the *location* of a resource.
A *URI* (uniform resource identifier) is a URL *or* a URN. So, to fully understand what URI means, you need to first understand what is a URN.
URN is an acronym for *uniform resource name*. There are may "unique identifier" schemes in the world, for example, ISBNs (globally unique for books), social security numbers (unique within a country), customer numbers (unique within a company's customers database) and telephone numbers. Each "unique identifier" scheme has its own notation. A URN is a wrapper for different "unique identifier" schemes. The syntax of a URN is "urn:<scheme-name>:<unique-identifier>". A URN uniquely identifies a *resource*, such as a book, person or piece of equipment. By itself, a URN does not specify the *location* of the resource. Instead, it is assumed that a *registry* provides a mapping from a resource's URN to its location. The URN specification does not state what form a registry takes, but it might be a database, a server application, a wall chart or anything else that is convenient. Some hypothetical examples of URNs are "urn:employee:08765245", "urn:customer:uk:3458:hul8" and "urn:foo:0000-0000-9E59-0000-5E-2". The <scheme-name> ("employee", "customer" and "foo" in these examples) part of a URN implicitly defines how to parse and interpret the <unique-identifier> that follows it. An arbitrary URN is meaningless unless: (1) you know the semantics implied by the <scheme-name>, and (2) you have access to the registry appropriate for the <scheme-name>. A registry does not have to be public or globally accessible. For example, "urn:employee:08765245" might be meaningful only within a specific company.
To date, URNs are not (yet) as popular as URLs. For this reason, URI is widely misused as a synonym for URL.
IRI is an acronym for *internationalized resource identifier*. An IRI is simply an internationalized version of a URI. In particular, a URI can contain letters and digits in the US-ASCII character set, while a IRI can contain those same letters and digits, and *also* European accented characters, Greek letters, Chinese ideograms and so on.

Components

Component is confusing terminology; *EndpointFactory* would have been more appropriate because a `Component` is a factory for creating `Endpoint` instances. For example, if a Camel-based application uses several JMS queues then the application will create one instance of the `JmsComponent` class (which implements the `Component` interface), and then the application invokes the `createEndpoint()` operation on this `JmsComponent` object several times. Each invocation of `JmsComponent.createEndpoint()` creates an instance of the `JmsEndpoint` class (which implements the `Endpoint` interface). Actually, application-level code does not invoke `Component.createEndpoint()` directly. Instead, application-level code normally invokes `CamelContext.getEndpoint()`; internally, the `CamelContext` object finds the desired `Component` object (as I will discuss shortly) and then invokes `createEndpoint()` on it.
Consider the following code.

```
myCamelContext.getEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword");
```

The parameter to `getEndpoint()` is a URI. The URI *prefix* (that is, the part before ":") specifies the name of a component. Internally, the `CamelContext` object maintains a mapping from names of components to `Component` objects. For the URI given in the above example, the `CamelContext` object would probably map the `pop3` prefix to an instance of the `MailComponent` class. Then the `CamelContext` object invokes `createEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword")` on that `MailComponent` object. The `createEndpoint()` operation splits the URI into its component parts and uses these parts to create and configure an `Endpoint` object.
In the previous paragraph, I mentioned that a `CamelContext` object maintains a mapping from component names to `Component` objects. This raises the question of how this map is populated with named `Component` objects. There are two ways of populating the map. The first way is for application-level code to invoke `CamelContext.addComponent(String componentName, Component component)`. The example below shows a single `MailComponent` object being registered in the map under 3 different names.

```
Component mailComponent = new org.apache.camel.component.mail.MailComponent();
myCamelContext.addComponent("pop3", mailComponent);
myCamelContext.addComponent("imap", mailComponent);
myCamelContext.addComponent("smtp", mailComponent);
```

The second (and preferred) way to populate the map of named `Component` objects in the `CamelContext` object is to let the `CamelContext` object perform lazy initialization. This approach relies on developers following a convention when they write a class that implements the `Component` interface. I illustrate the convention by an example. Let's assume you write a class called `com.example.myproject.FooComponent` and you want Camel to automatically recognize this by the name "foo". To do this, you have to write a properties file called "META-INF/services/org/apache/camel/component/foo" (without a ".properties" file extension) that has a single entry in it called `class`, the value of which is the fully-scoped name of your class. This is shown below.

Listing 1. META-INF/services/org/apache/camel/component/foo

```
class=com.example.myproject.FooComponent
```

If you want Camel to also recognize the class by the name "bar" then you write another properties file in the same directory called "bar" that has the same contents. Once you have written the properties file(s), you create a jar file that contains the `com.example.myproject.FooComponent` class and the properties file(s), and you add this jar file to your CLASSPATH. Then, when application-level code invokes `createEndpoint("foo:...")` on a `CamelContext` object, Camel will find the "foo"" properties file on the CLASSPATH, get the value of the `class` property from that properties file, and use reflection APIs to create an instance of the specified class.
As I said in Section 4.1 ("Endpoint"), Camel provides out-of-the-box support for numerous communication technologies. The out-of-the-box support consists of classes that implement the `Component` interface plus properties files that enable a `CamelContext` object to populate its map of named `Component` objects.
Earlier in this section I gave the following example of calling `CamelContext.getEndpoint()`.

```
myCamelContext.getEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword");
```

When I originally gave that example, I said that the parameter to `getEndpoint()` was a URI. I said that because the online Camel documentation and the Camel source code both claim the parameter is a URI. In reality, the parameter is restricted to being a URL. This is because when Camel extracts the component name from the parameter, it looks for the first ":", which is a simplistic algorithm. To understand why, recall from Section 4.4 ("The Meaning of URL, URI, URN and IRI") that a URI can be a URL *or* a URN. Now consider the following calls to `getEndpoint`.

```
myCamelContext.getEndpoint("pop3:...");
myCamelContext.getEndpoint("jms:...");
myCamelContext.getEndpoint("urn:foo:...");
myCamelContext.getEndpoint("urn:bar:...");
```

Camel identifies the components in the above example as "pop3", "jms", "urn" and "urn". It would be more useful if the latter components were identified as "urn:foo" and "urn:bar" or, alternatively, as "foo" and "bar" (that is, by skipping over the "urn:" prefix). So, in practice you must identify an endpoint with a URL (a string of the form "<scheme>:...") rather than with a URN (a string of the form "urn:<scheme>:..."). This lack of proper support for URNs means the you should consider the parameter to `getEndpoint()` as being a URL rather than (as claimed) a URI.

Message and Exchange

The `Message` interface provides an abstraction for a single message, such as a request, reply or exception message.
There are concrete classes that implement the `Message` interface for each Camel-supported communications technology. For example, the `JmsMessage` class provides a JMS-specific implementation of the `Message` interface. The public API of the `Message` interface provides get- and set-style methods to access the *message id*, *body* and individual *header* fields of a messge.
The `Exchange` interface provides an abstraction for an exchange of messages, that is, a request message and its corresponding reply or exception message. In Camel terminology, the request, reply and exception messages are called *in*, *out* and *fault* messages.
There are concrete classes that implement the `Exchange` interface for each Camel-supported communications technology. For example, the `JmsExchange` class provides a JMS-specific implementation of the `Exchange` interface. The public API of the `Exchange` interface is quite limited. This is intentional, and it is expected that each class that implements this interface will provide its own technology-specific operations.
Application-level programmers rarely access the `Exchange` interface (or classes that implement it) directly. However, many classes in Camel are generic types that are instantiated on (a class that implements) `Exchange`. Because of this, the `Exchange` interface appears a lot in the generic signatures of classes and methods.

Processor

The `Processor` interface represents a class that processes a message. The signature of this interface is shown below.

Listing 1. Processor

```
package org.apache.camel;
public interface Processor {
    void process(Exchange exchange) throws Exception;
}
```

Notice that the parameter to the `process()` method is an `Exchange` rather than a `Message`. This provides flexibility. For example, an implementation of this method initially might call `exchange.getIn()` to get the input

message and process it. If an error occurs during processing then the method can call `exchange.setException()`.
An application-level developer might implement the `Processor` interface with a class that executes some business logic. However, there are many classes in the Camel library that implement the `Processor` interface in a way that provides support for a design pattern in the EIP book. For example, `ChoiceProcessor` implements the message router pattern, that is, it uses a cascading if-then-else statement to route a message from an input queue to one of several output queues. Another example is the `FilterProcessor` class which discards messages that do not satisfy a stated *predicate* (that is, condition).

Routes, RouteBuilders and Java DSL

A *route* is the step-by-step movement of a `Message` from an input queue, through arbitrary types of decision making (such as filters and routers) to a destination queue (if any). Camel provides two ways for an application developer to specify routes. One way is to specify route information in an XML file. A discussion of that approach is outside the scope of this document. The other way is through what Camel calls a Java *DSL* (domain-specific language).

Introduction to Java DSL

For many people, the term "domain-specific language" implies a compiler or interpreter that can process an input file containing keywords and syntax specific to a particular domain. This is *not* the approach taken by Camel. Camel documentation consistently uses the term "Java DSL" instead of "DSL", but this does not entirely avoid potential confusion. The Camel "Java DSL" is a class library that can be used in a way that looks almost like a DSL, except that it has a bit of Java syntactic baggage. You can see this in the example below. Comments afterwards explain some of the constructs used in the example.

Listing 1. Example of Camel's "Java DSL"

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").filter(header("foo").isEqualTo("bar")).to("queue:b");
        from("queue:c").choice()
                .when(header("foo").isEqualTo("bar")).to("queue:d")
                .when(header("foo").isEqualTo("cheese")).to("queue:e")
                .otherwise().to("queue:f");
    }
};
CamelContext myCamelContext = new DefaultCamelContext();
myCamelContext.addRoutes(builder);
```

The first line in the above example creates an object which is an instance of an anonymous subclass of `RouteBuilder` with the specified `configure()` method.
The `CamelContext.addRoutes(RouterBuilder builder)` method invokes `builder.setContext(this)` – so the `RouteBuilder` object knows which `CamelContext` object it is associated with – and then invokes `builder.configure()`. The body of `configure()` invokes methods such as `from()`, `filter()`, `choice()`, `when()`, `isEqualTo()`, `otherwise()` and `to()`.
The `RouteBuilder.from(String uri)` method invokes `getEndpoint(uri)` on the `CamelContext` associated with the `RouteBuilder` object to get the specified `Endpoint` and then puts a `FromBuilder` "wrapper" around this `Endpoint`. The `FromBuilder.filter(Predicate predicate)` method creates a `FilterProcessor` object for the `Predicate` (that is, condition) object built from the `header("foo").isEqualTo("bar")` expression. In this way, these operations incrementally build up a `Route` object (with a `RouteBuilder` wrapper around it) and add it to the `CamelContext` object associated with the `RouteBuilder`.

Critique of Java DSL

The online Camel documentation compares Java DSL favourably against the alternative of configuring routes and endpoints in a XML-based Spring configuration file. In particular, Java DSL is less verbose than its XML counterpart. In addition, many integrated development environments (IDEs) provide an auto-completion feature in their editors. This auto-completion feature works with Java DSL, thereby making it easier for developers to write Java DSL.
However, there is another option that the Camel documentation neglects to consider: that of writing a parser that can process DSL stored in, say, an external file. Currently, Camel does not provide such a DSL parser, and I do not know if it is on the "to do" list of the Camel maintainers. I think that a DSL parser would offer a significant benefit over the current Java DSL. In particular, the DSL would have a syntactic definition that could be expressed in a relatively short BNF form. The effort required by a Camel user to learn how to use DSL by reading this BNF would almost certainly be significantly less than the effort currently required to study the API of the `RouterBuilder` classes.

Continue Learning about Camel

Return to the main Getting Started page for additional introductory reference information.

CHAPTER 4

Architecture

Camel uses a Java based Routing Domain Specific Language (DSL) or an Xml Configuration to configure routing and mediation rules which are added to a CamelContext to implement the various Enterprise Integration Patterns.
At a high level Camel consists of a CamelContext which contains a collection of Component instances. A Component is essentially a factory of Endpoint instances. You can explicitly configure Component instances in Java code or an IoC container like Spring or Guice, or they can be auto-discovered using URIs.

An Endpoint acts rather like a URI or URL in a web application or a Destination in a JMS system; you can communicate with an endpoint; either sending messages to it or consuming messages from it. You can then create a Producer or Consumer on an Endpoint to exchange messages with it.

The DSL makes heavy use of pluggable Languages to create an Expression or Predicate to make a truly powerful DSL which is extensible to the most suitable language depending on your needs. The following languages are supported

- Bean Language for using Java for expressions
- Constant
- the unified EL from JSP and JSF
- Header
- JSonPath
- JXPath
- Mvel
- OGNL
- Ref Language
- ExchangeProperty / Property
- Scripting Languages such as
  - BeanShell
  - JavaScript
  - Groovy
  - Python
  - PHP
  - Ruby
- Simple
  - File Language
- Spring Expression Language
- SQL
- Tokenizer
- XPath
- XQuery
- VTD-XML

Most of these languages is also supported used as Annotation Based Expression Language.

For a full details of the individual languages see the Language Appendix

URIS

Camel makes extensive use of URIs to allow you to refer to endpoints which are lazily created by a Component if you refer to them within Routes.
important
Make sure to read How do I configure endpoints to learn more about configuring endpoints. For example how to refer to beans in the Registry or how to use raw values for password options, and using property placeholders etc.

Current Supported URIs

Component / ArtifactId / URI	Description
AHC / `camel-ahc` `ahc:http[s]://hostName[:port][/resourceUri][?options]`	To call external HTTP services using Async Http Client
AHC-WS / `camel-ahc-ws`	To exchange data with external Websocket servers using

`ahc-ws[s]://hostName[:port][/resourceUri][?options]`	Async Http Client
AMQP / `camel-amqp` `amqp:[queue:|topic:]destinationName[?options]`	For Messaging with AMQP protocol
APNS / `camel-apns` `apns:<notify|consumer>[?options]`	For sending notifications to Apple iOS devices
Atmosphere-Websocket / `camel-atmosphere-websocket` `atmosphere-websocket:///relative path[?options]`	To exchange data with external Websocket clients using Atmosphere
Atom / `camel-atom` `atom:atomUri[?options]`	Working with Apache Abdera for atom integration, such as consuming an atom feed.
Avro / `camel-avro` `avro:[transport]:[host]:[port][/messageName][?options]`	Working with Apache Avro for data serialization.
AWS-CW / camel-aws `aws-cw://namespace[?options]`	For working with Amazon's CloudWatch (CW).
AWS-DDB / camel-aws `aws-ddb://tableName[?options]`	For working with Amazon's DynamoDB (DDB).
AWS-DDBSTREAM / camel-aws `aws-ddbstream://tableName[?options]`	For working with Amazon's DynamoDB Streams (DDB Streams).
AWS-EC2 / camel-aws `aws-ec2://label[?options]`	For working with Amazon's Elastic Compute Cloud (EC2).
AWS-SDB / camel-aws `aws-sdb://domainName[?options]`	For working with Amazon's SimpleDB (SDB).
AWS-SES / camel-aws `aws-ses://from[?options]`	For working with Amazon's Simple Email Service (SES).
AWS-SNS / camel-aws `aws-sns://topicName[?options]`	For Messaging with Amazon's Simple Notification Service (SNS).
AWS-SQS / camel-aws `aws-sqs://queueName[?options]`	For Messaging with Amazon's Simple Queue Service (SQS).
AWS-SWF / camel-aws `aws-swf://<worfklow|activity>[?options]`	For Messaging with Amazon's Simple Workflow Service (SWF).
AWS-S3 / camel-aws `aws-s3://bucketName[?options]`	For working with Amazon's Simple Storage Service (S3).

Bean / `camel-core` `bean:beanName[?options]`	Uses the Bean Binding to bind message exchanges to beans in the Registry. Is also used for exposing and invoking POJO (Plain Old Java Objects).
Beanstalk / `camel-beanstalk` `beanstalk:hostname:port/tube[?options]`	For working with Amazon's Beanstalk.
Bean Validator / `camel-bean-validator` `bean-validator:label[?options]`	Validates the payload of a message using the Java Validation API (JSR 303 and JAXP Validation) and its reference implementation Hibernate Validator
Box / `camel-box` `box://endpoint-prefix/endpoint?[options]`	For uploading, downloading and managing files, managing files, folders, groups, collaborations, etc. on Box.com.
Braintree / `camel-braintree` `braintree://endpoint-prefix/endpoint?[options]`	Component for interacting with Braintree Payments via Braintree Java SDK
Browse / `camel-core` `browse:someName`	Provides a simple BrowsableEndpoint which can be useful for testing, visualisation tools or debugging. The exchanges sent to the endpoint are all available to be browsed.
Cache / `camel-cache` `cache://cacheName[?options]`	The cache component facilitates creation of caching endpoints and processors using EHCache as the cache implementation.
Cassandra / `camel-cassandraql` `cql:localhost/keyspace`	For integrating with Apache Cassandra.
Class / `camel-core` `class:className[?options]`	Uses the Bean Binding to bind message exchanges to beans in the Registry. Is also used for exposing and invoking POJO (Plain Old Java Objects).
Chronicle Engine / `camel-chronicle` `chronicle-engine:addresses/path[?options]`	Chronicle Engine is a high performance, low latency, reactive processing framework.
Chunk / `camel-chunk` `chunk:templateName[?options]`	Generates a response using a Chunk template
CMIS / `camel-cmis` `cmis://cmisServerUrl[?options]`	Uses the Apache Chemistry client API to interface with CMIS supporting CMS
Cometd / `camel-cometd` `cometd://hostName:port/channelName[?options]`	Used to deliver messages using the jetty cometd implementation of the bayeux protocol
Consul / `camel-consul` `consul:apiEndpoint[?options]`	For interfacing with an Consul.
Context / `camel-context` `context:camelContextId:localEndpointName[?options]`	Used to refer to endpoints within a separate CamelContext to provide a simple black box composition approach so that routes can be combined into a CamelContext and then used as a black box component inside other routes in other CamelContexts
ControlBus / `camel-core`	ControlBus EIP that allows to send messages to Endpoints for managing and monitoring your Camel applications.

`controlbus:command[?options]`	
CouchDB / `camel-couchdb` `couchdb:hostName[:port]/database[?options]`	To integrate with Apache CouchDB.
Crypto (Digital Signatures) / `camel-crypto` `crypto:<sign|verify>:name[?options]`	Used to sign and verify exchanges using the Signature Service of the Java Cryptographic Extension.
CXF / `camel-cxf` `cxf:<bean:cxfEndpoint|//someAddress>[?options]`	Working with Apache CXF for web services integration
CXF Bean / `camel-cxf` `cxfbean:serviceBeanRef[?options]`	Proceess the exchange using a JAX WS or JAX RS annotated bean from the registry. Requires less configuration than the above CXF Component
CXFRS / `camel-cxf` `cxfrs:<bean:rsEndpoint|//address>[?options]`	Working with Apache CXF for REST services integration
DataFormat / `camel-core` `dataformat:name:<marshal|unmarshal>[?options]`	for working with Data Formats as if it was a regular Component supporting Endpoints and URIs.
DataSet / `camel-core` `dataset:name[?options]`	For load & soak testing the DataSet provides a way to create huge numbers of messages for sending to Components or asserting that they are consumed correctly
Direct / `camel-core` `direct:someName[?options]`	Synchronous call to another endpoint from **same** CamelContext.
Direct-VM / `camel-core` `direct-vm:someName[?options]`	Synchronous call to another endpoint in another CamelContext running in the same JVM.
DNS / `camel-dns` `dns:operation[?options]`	To lookup domain information and run DNS queries using DNSJava
Disruptor / `camel-disruptor` `disruptor:someName[?<option>]` `disruptor-vm:someName[?<option>]`	To provide the implementation of SEDA which is based on disruptor
Docker / `camel-docker` `docker://[operation]?[options]`	To communicate with Docker
Dozer / `camel-dozer` `dozer://name?[options]`	To convert message body using the Dozer type converter library.
Dropbox / `camel-dropbox` `dropbox://[operation]?[options]`	The **dropbox:** component allows you to treat Dropbox remote folders as a producer or consumer of messages.
EJB / `camel-ejb` `ejb:ejbName[?options]`	Uses the Bean Binding to bind message exchanges to EJBs. It works like the Bean component but just for accessing EJBs. Supports EJB 3.0 onwards.
Ehcache / `camel-ehcache`	The cache component facilitates creation of caching endpoints and processors using Ehcache 3 as the cache

`ehcache://cacheName[?options]`	implementation.
ElasticSearch / `camel-elasticsearch` `elasticsearch://clusterName[?options]`	For interfacing with an ElasticSearch server.
Etcd / `camel-etcd` `etcd:namespace[/path][?options]`	For interfacing with an Etcd key value store.
Spring Event / `camel-spring` `spring-event://default`	Working with Spring ApplicationEvents
EventAdmin / `camel-eventadmin` `eventadmin:topic[?options]`	Receiving OSGi EventAdmin events
Exec / `camel-exec` `exec://executable[?options]`	For executing system commands
Facebook / `camel-facebook` `facebook://endpoint[?options]`	Providing access to all of the Facebook APIs accessible using Facebook4J
File / `camel-core` `file://nameOfFileOrDirectory[?options]`	Sending messages to a file or polling a file or directory.
Flatpack / `camel-flatpack` `flatpack:[fixed|delim]:configFile[?options]`	Processing fixed width or delimited files or messages using the FlatPack library

Flink / `camel-flink`
`flink:dataset[?options]`
`flink:datastream[?options]`
	Bridges Camel connectors with Apache Flink tasks.

FOP / `camel-fop` `fop:outputFormat[?options]`	Renders the message into different output formats using Apache FOP
FreeMarker / `camel-freemarker` `freemarker:templateName[?options]`	Generates a response using a FreeMarker template
FTP / `camel-ftp` `ftp:contextPath[?options]`	Sending and receiving files over FTP.
FTPS / `camel-ftp` `ftps://[username@]hostName[:port]/directoryName[?options]`	Sending and receiving files over FTP Secure (TLS and SSL).
Ganglia / `camel-ganglia` `ganglia:destination:port[?options]`	Sends values as metrics to the Ganglia performance monitoring system using gmetric4j. Can be used along with JMXetric.
GAuth / camel-gae `gauth://name[?options]`	Used by web applications to implement an OAuth consumer. See also Camel Components for Google App Engine.
GHttp / camel-gae	Provides connectivity to the URL fetch service of Google App Engine but can also be used to receive messages from

`ghttp:contextPath[?options]`	servlets. See also Camel Components for Google App Engine.
Git / camel-git `git:localRepositoryPath[?options]`	Supports interaction with Git repositories
Github / camel-github `github:endpoint[?options]`	Supports interaction with Github
GLogin / camel-gae `glogin://hostname[:port][?options]`	Used by Camel applications outside Google App Engine (GAE) for programmatic login to GAE applications. See also Camel Components for Google App Engine.
GTask / camel-gae `gtask://queue-name[?options]`	Supports asynchronous message processing on Google App Engine by using the task queueing service as message queue. See also Camel Components for Google App Engine.
Google Calendar / camel-google-calendar `google-calendar://endpoint-prefix/endpoint?[options]`	Supports interaction with Google Calendar's REST API.
Google Drive / camel-google-drive `google-drive://endpoint-prefix/endpoint?[options]`	Supports interaction with Google Drive's REST API.
Google Mail / camel-google-mail `google-mail://endpoint-prefix/endpoint?[options]`	Supports interaction with Google Mail's REST API.
GMail / camel-gae `gmail://user@g[oogle]mail.com[?options]`	Supports sending of emails via the mail service of Google App Engine. See also Camel Components for Google App Engine.
Gora / `camel-gora` `gora:instanceName[?options]`	Supports to work with NoSQL databases using the Apache Gora framework.
Grape/ `camel-grape` `grape:defaultMavenCoordinates`	Grape component allows you to fetch, load and manage additional jars when CamelContext is running.
Geocoder / `camel-geocoder` `geocoder:<address|latlng:latitude,longitude>[?options]`	Supports looking up geocoders for an address, or reverse lookup geocoders from an address.
Google Guava EventBus / `camel-guava-eventbus` `guava-eventbus:busName[?options]`	The Google Guava EventBus allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other). This component provides integration bridge between Camel and Google Guava EventBus infrastructure.
Hazelcast / camel-hazelcast `hazelcast://[type]:cachename[?options]`	Hazelcast is a data grid entirely implemented in Java (single jar). This component supports map, multimap, seda, queue, set, atomic number and simple cluster support.
HBase / `camel-hbase` `hbase://table[?options]`	For reading/writing from/to an HBase store (Hadoop database)
HDFS / `camel-hdfs` `hdfs://hostName[:port][/path][?options]`	For reading/writing from/to an HDFS filesystem using Hadoop 1.x

HDFS2 / `camel-hdfs2` `hdfs2://hostName[:port][/path][?options]`	For reading/writing from/to an HDFS filesystem using Hadoop 2.x
Hipchat / `camel-hipchat` `hipchat://[host][:port]?options`	For sending/receiving messages to Hipchat using v2 API
HL7 / `camel-hl7` `mina2:tcp://hostName[:port][?options]`	For working with the HL7 MLLP protocol and the HL7 data format using the HAPI library
Infinispan / `camel-infinispan` `infinispan://cacheName[?options]`	For reading/writing from/to Infinispan distributed key/value store and data grid
HTTP / `camel-http` `http:hostName[:port][/resourceUri][?options]`	For calling out to external HTTP servers using Apache HTTP Client 3.x
HTTP4 / `camel-http4` `http4:hostName[:port][/resourceUri][?options]`	For calling out to external HTTP servers using Apache HTTP Client 4.x
iBATIS / `camel-ibatis` `ibatis://statementName[?options]`	Performs a query, poll, insert, update or delete in a relational database using Apache iBATIS
Ignite / `camel-ignite` `ignite:[cache/compute/messaging/...][?options]`	Apache Ignite In-Memory Data Fabric is a high-performance, integrated and distributed in-memory platform for computing and transacting on large-scale data sets in real-time, orders of magnitude faster than possible with traditional disk-based or flash technologies. It is designed to deliver uncompromised performance for a wide set of in-memory computing use cases from high performance computing, to the industry most advanced data grid, highly available service grid, and streaming.
IMAP / `camel-mail` `imap://[username@]hostName[:port][?options]`	Receiving email using IMAP
IMAPS / `camel-mail` `imaps://[username@]hostName[:port][?options]`	...
IRC / `camel-irc` `irc:[login@]hostName[:port]/#room[?options]`	For IRC communication
IronMQ / `camel-ironmq` `ironmq:queueName[?options]`	For working with IronMQ a elastic and durable hosted message queue as a service.
JavaSpace / `camel-javaspace` `javaspace:jini://hostName[?options]`	Sending and receiving messages through JavaSpace
jBPM / `camel-jbpm` `jbpm:hostName[:port][/resourceUri][?options]`	Sending messages through kie-remote-client API to jBPM.
jcache / `camel-jcache` `jcache:cacheName[?options]`	The JCache component facilitates creation of caching endpoints and processors using JCache / jsr107 as the cache implementation.
jclouds / `camel-jclouds`	For interacting with cloud compute & blobstore service via

`jclouds:<blobstore|compute>:[provider id][?options]`	jclouds
JCR / `camel-jcr` `jcr://user:password@repository/path/to/node[?options]`	Storing a message in a JCR compliant repository like Apache Jackrabbit
JDBC / `camel-jdbc` `jdbc:dataSourceName[?options]`	For performing JDBC queries and operations
Jetty / `camel-jetty` `jetty:hostName[:port][/resourceUri][?options]`	For exposing or consuming services over HTTP
JGroups / `camel-jgroups` `jgroups:clusterName[?options]`	The `jgroups:` component provides exchange of messages between Camel infrastructure and JGroups clusters.
JIRA / `camel-jira` `jira://endpoint[?options]`	For interacting with JIRA
JMS / `camel-jms` `jms:[queue:|topic:]destinationName[?options]`	Working with JMS providers
JMX / `camel-jmx` `jmx://platform[?options]`	For working with JMX notification listeners
JPA / `camel-jpa` `jpa://entityName[?options]`	For using a database as a queue via the JPA specification for working with OpenJPA, Hibernate or TopLink
JOLT / `camel-jolt` `jolt:specName[?options]`	The **jolt:** component allows you to process a JSON messages using an JOLT specification. This can be ideal when doing JSON to JSON transformation.
Jsch / `camel-jsch` `scp://hostName[:port]/destination[?options]`	Support for the scp protocol
JT/400 / `camel-jt400` `jt400://user:pwd@system/<path_to_dtaq>[?options]`	For integrating with data queues on an AS/400 (aka System i, IBM i, i5, ...) system
Kafka / `camel-kafka` `kafka://server:port[?options]`	For producing to or consuming from Apache Kafka message brokers.
Kestrel / `camel-kestrel` `kestrel://[addresslist/]queueName[?options]`	For producing to or consuming from Kestrel queues
Krati / `camel-krati` `krati://[path to datastore/][?options]`	For producing to or consuming to Krati datastores
Kubernetes / `camel-kubernetes` `kubernetes:masterUrl[?options]`	For integrating your application with Kubernetes standalone or on top of OpenShift.

Kura / `camel-kura`	For deploying Camel OSGi routes into the Eclipse Kura M2M container.
Language / `camel-core` `language://languageName[:script][?options]`	Executes Languages scripts
LDAP / `camel-ldap` `ldap:host[:port][?options]`	Performing searches on LDAP servers (<scope> must be one of object|onelevel|subtree)
LinkedIn / `camel-linkedin` `linkedin://endpoint-prefix/endpoint?[options]`	Component for retrieving LinkedIn user profiles, connections, companies, groups, posts, etc. using LinkedIn REST API.
Log / `camel-core` `log:loggingCategory[?options]`	Uses Jakarta Commons Logging to log the message exchange to some underlying logging system like log4j
Lucene / `camel-lucene` `lucene:searcherName:<insert|query>[?options]`	Uses Apache Lucene to perform Java-based indexing and full text based searches using advanced analysis/tokenization capabilities
Lumberjack / `camel-lumberjack` `lumberjack:host[:port]`	Uses the Lumberjack protocol for retrieving logs (from Filebeat for instance)
Metrics / `camel-metrics` `metrics:[meter|counter|histogram|timer]:metricname[?options]`	Uses Metrics to collect application statistics directly from Camel routes.
MINA / `camel-mina` `mina:[tcp|udp|vm]:host[:port][?options]`	Working with Apache MINA 1.x
MINA2 / `camel-mina2` `mina2:[tcp|udp|vm]:host[:port][?options]`	Working with Apache MINA 2.x
Mock / `camel-core` `mock:name[?options]`	For testing routes and mediation rules using mocks
MLLP / `camel-mllp` `mllp:host:port[?options]`	The MLLP component is specifically designed to handle the nuances of the MLLP protocol and provide the functionality required by Healthcare providers to communicate with other systems using the MLLP protocol
MongoDB / `camel-mongodb` `mongodb:connectionBean[?options]`	Interacts with MongoDB databases and collections. Offers producer endpoints to perform CRUD-style operations and more against databases and collections, as well as consumer endpoints to listen on collections and dispatch objects to Camel routes
MongoDB GridFS / `camel-mongodb-gridfs` `mongodb-gridfs:dbName[?options]`	Sending and receiving files via MongoDB's GridFS system. **Note:** for Camel < 2.19, the URI syntax is gridfs:dbName[?options]
MQTT / `camel-mqtt` `mqtt:name[?options]`	Component for communicating with MQTT M2M message brokers
MSV / `camel-msv` `msv:someLocalOrRemoteResource[?options]`	Validates the payload of a message using the MSV Library