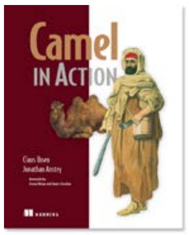
MANNING PUBLICATIONS



Camel in Action
Claus Ibsen, Jonathan Anstey, and Hadrian Zbarcea

In this article, based on chapter 2 of Camel in Action, the authors show how to design and implement solutions to enterprise integration problems using EIPs and Camel.

You may also be interested in…

Routing Using Camel's Implementation of the EIPs

One of the most important features of Camel is routing; without it, Camel would essentially be a library of transport connectors. In this article, we'll dive into routing with Camel.

Routing happens in many aspects of everyday life. When you mail a letter, for instance, it may be routed through several cities before reaching its final address. An email you send will be routed through many different computer network systems before reaching its final destination. In all cases, the router's function is to selectively move the message forward.

In the context of enterprise messaging systems, routing is the process whereby a message is taken from an input queue and, based on a set of conditions, is sent to one of several output queues, as shown in figure 1. This effectively means that the input and output queues are unaware of the conditions in between them. The conditional logic is decoupled from the message consumer and producer.

In Apache Camel, routing is a more general concept. It's defined as a step-by-step movement of the message, which originates from an endpoint in the role of a consumer. The consumer could be receiving the message from an external service, polling for the message on some system, or even creating the message itself. This message then flows through a processing component, which could be an enterprise integration pattern (EIP), a processor, an interceptor, or some other custom creation. The message is finally sent to a target endpoint that is in the role of a producer. A route may have many processing components that modify the message or send it to another location or it may have none, in which case it would be a simple pipeline.

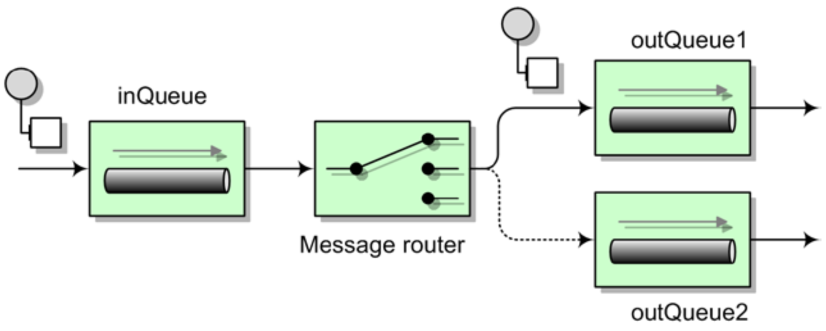


Figure 1 A message router consumes messages from an input channel and, depending on a set of conditions, sends the message to one of a set of output channels.

In this article, we'll use a fictional company. We will give you a glimpse of how to design and implement solutions to enterprise integration problems using EIPs and Camel. By the end of the article, you'll be proficient enough to create useful routing applications with Camel.

Introducing Rider Auto Parts

Our fictional motorcycle parts business, Rider Auto Parts, supplies parts to motorcycle manufacturers. Over the years, they've changed the way they receive orders several times. Initially, orders were placed by uploading comma-separated value (CSV) files to an FTP server. The message format was later changed to XML. Currently they provide a web site through which orders are submitted as XML messages over HTTP.

Rider Auto Parts asks new customers to use the web interface to place orders but, because of service level agreements (SLAs) with existing customers, they must keep all the old message formats and interfaces up and running. All of these messages are converted to an internal Plain Old Java Object (POJO) format before processing. A high-level view of the order processing system is shown in figure 2.

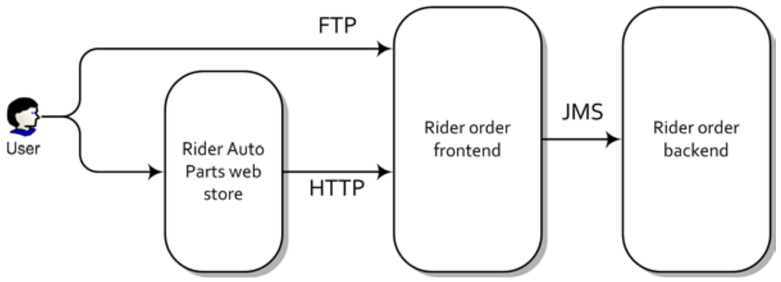


Figure 2 A customer has two ways of submitting orders to the Rider Auto Parts order handling system: either by uploading the raw order file to an FTP server or by submitting an order through the Rider Auto Parts web store. All orders are eventually sent via JMS for processing at Rider Auto Parts.

Rider Auto Parts faces a pretty common problem: over years of operation, they have acquired software baggage in the form of transports and data formats that were popular at the time. This is no problem for an integration framework like Camel, though. The goal is to help Rider Auto Parts implement their current requirements and new functionality using Camel.

First, we would implement the FTP module of the order system. This would involve:

1. Polling the FTP server and downloading new orders
2. Converting the order files to JMS Messages
3. Sending the messages to the JMS incomingOrders queue

To complete steps 1 and 3, we need to understand how to communicate over FTP and JMS using Camel's endpoints. To complete the entire assignment, we would need to understand routing with the Java DSL.

Our next order of business is to tackle routing using Camel's implementation of the EIPs.

Routing and EIPs

As far as EIPs go, we'll be looking at the Content Based Router, Message Filter, Multicast, Recipient List, and Wire Tap.

Using a content-based router

As the name implies, a Content-Based Router (CBR) is a message router that routes a message to a destination based on its content. The content could be a message header, the payload data type, part of the payload itself—pretty much anything in the message exchange.

Let's cue in Rider Auto Parts. Some customers have started uploading orders to the FTP server in the newer XML format rather than CSV. That means we have two types of messages coming into the incomingOrders queue. We need to convert the incoming orders into an internal POJO format. We obviously need to do different conversion for the different types of incoming orders.

As a possible solution, we could use the filename extension to determine whether to send a particular order message to a queue for CSV orders or a queue for XML orders. This is depicted in figure 3.

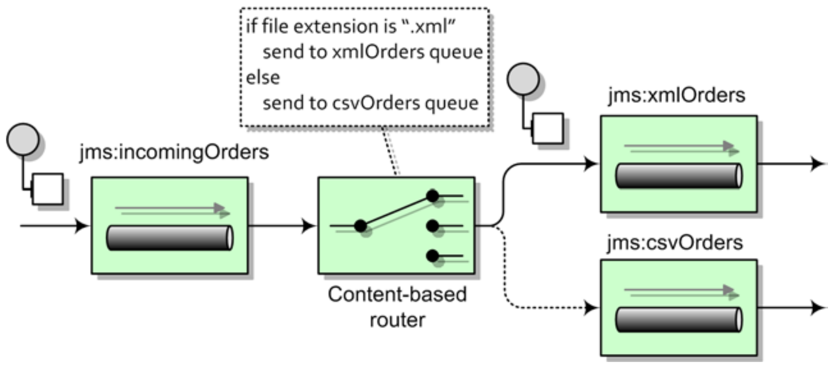


Figure 3 The Content-Based Router routes messages based on their content. In this case, the filename extension (as a message header) is used to determine which queue to route to.

We can use the CamelFileName header set by the FTP consumer to get the filename.

To do the conditional routing required by the Content-Based Router, Camel introduces a few keywords in the DSL. The `choice` method creates a content-based router processor, and conditions are added by following `choice` with a combination of a `when` method and a predicate.

We could have chosen `contentBasedRouter` for the method name to match the EIP but we stuck with `choice` because it reads more naturally. It looks like this:

```
from("jms:incomingOrders")
.choice()
    .when(predicate)
        .to("jms:xmlOrders")
    .when(predicate)
        .to("jms:csvOrders");
```

You may have noticed that we didn't fill in the predicates required for each `when` method. A predicate in Camel is a simple interface that only has a `matches` method:

```
public interface Predicate {
    boolean matches(Exchange exchange);
}
```

For example, you can think of a predicate as a boolean condition in a Java if statement.

You probably don't want to look inside the Exchange yourself and do a comparison. Fortunately, predicates are often built up from expressions, and expressions are used to extract a result from an exchange based on the expression content. There are many different expression languages to choose from in Camel, some of which include EL, JXPath, Mvel, OGNL, PHP, BeanShell, JavaScript, Groovy, Python, Ruby, XPath, and XQuery. You can even use a method call to a bean as an expression in Camel. In our case, we'll be using the expression builder methods that are part of the Java DSL.

Within the RouteBuilder, we can start by using the `header` method, which returns an expression that will evaluate to the header value. For example, `header("CamelFileName")` will create an Expression that will resolve to the value of the CamelFileName header on the incoming Exchange. On this Expression we can invoke a number of methods to create a predicate. So, to check if the filename extension is equal to ".xml", we can use the following predicate:

```
header("CamelFileName").endsWith(".xml")
```

Our completed content-based router is shown in listing 1.

Listing 1 A complete content-based router

```
context.addRoutes(new RouteBuilder() {
    public void configure() {
        from("file:src/data?noop=true").to("jms:incomingOrders");

        from("jms:incomingOrders")                              #1
        .choice()                                               #1
            .when(header("CamelFileName").endsWith(".xml")) #1
                .to("jms:xmlOrders")                            #1
            .when(header("CamelFileName").endsWith(".csv")) #1
                .to("jms:csvOrders");                           #1

        from("jms:xmlOrders").process(new Processor() { #2
            public void process(Exchange exchange) throws Exception {
                System.out.println("Recieved XML order: "
                        + exchange.getIn().getHeader("CamelFileName"));
            }
        });

        from("jms:csvOrders").process(new Processor() { #2
            public void process(Exchange exchange) throws Exception {
                System.out.println("Recieved CSV order: "
                        + exchange.getIn().getHeader("CamelFileName"));
            }
        });
    }
});
```

#1 Content-based router
#2 Test routes that print message content

To run this example, go to the chapter2/cbr directory in Camel in Action's source and run this Maven command:

```
mvn compile exec:java -Dexec.mainClass=camelinaction.OrderRouter
```

This will consume two order files in the chapter2/cbr/src/data directory and output the following:

```
Received CSV order: message2.csv
Received XML order: message1.xml
```

The output comes from the two routes at the end of the `configure` method #2. These routes consume messages from the xmlOrders and csvOrders queues and then print out messages. We are using these routes to test that our router #1 is working as expected.

USING OTHERWISE

One of Rider Auto Parts' customers sends CSV orders with the ".csl" extension. Our current route only handles ".csv" and ".xml" files and will drop all orders with other extensions. This is not a good solution, so we need to improve things a bit.

One way to handle the extra extension is to use a regular expression as a predicate instead of the simple endsWith call. The following route can handle the extra file extension:

```
from("jms:incomingOrders")
.choice()
    .when(header("CamelFileName").endsWith(".xml"))
        .to("jms:xmlOrders")
    .when(header("CamelFileName").regex("^.*(csv|csl)$"))
        .to("jms:csvOrders");
```

This solution still suffers from the same problem though—any orders not conforming to the file extension scheme will be dropped. Really, we should be handling bad orders that come in so someone can fix the problem. For this we can use the "otherwise" clause:

```
from("jms:incomingOrders")
.choice()
    .when(header("CamelFileName").endsWith(".xml"))
```

```
            .to("jms:xmlOrders")
        .when(header("CamelFileName").regex("^.*(csv|csl)$"))
            .to("jms:csvOrders")
        .otherwise()
            .to("jms:badOrders");
```

Now, all orders not having an extension of ".csv", ".csl", or ".xml" are sent to the badOrders queue for handling. To run this example, go to the chapter2/cbr directory in Camel in Action's source and run this command:

```
mvn compile exec:java -Dexec.mainClass=camelinaction.OrderRouterOtherwise
```

This will consume four order files in the chapter2/cbr/src/data directory and output the following:

```
Received CSV order: message2.csv
Received XML order: message1.xml
Received bad order: message4.bad
Received CSV order: message3.csl
```

We can now see that a bad order has been received.

ROUTING AFTER A CBR

The content-based router may seem like it is the end of the route; messages are routed to one of several destinations, and that's it. Continuing the flow means you need another route right?

Well, there are several ways we can continue routing after a CBR. One is by using another route, like we did in listing 1 for printing a test message to the console. Another way of continuing the flow is by closing the choice "block" and adding another processor to the pipeline after that.

You can close the choice block by using the end method:

```
from("jms:incomingOrders")
.choice()
    .when(header("CamelFileName").endsWith(".xml"))
        .to("jms:xmlOrders")
    .when(header("CamelFileName").regex("^.*(csv|csl)$"))
        .to("jms:csvOrders")
    .otherwise()
        .to("jms:badOrders")
.end()
.to("jms:continuedProcessing");
```

Here, we've closed the choice and added another "to" to our route. Now, after each destination with the choice, the message will be routed to the "continuedProcessing" queue as well. This is illustrated in figure 4.

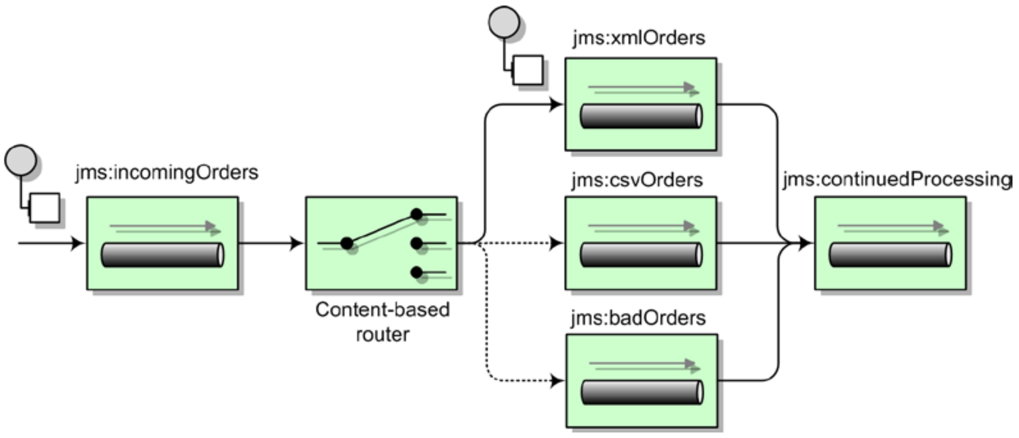


Figure 4 By using the end method, you can route messages to a destination after the Content-Based Router.

We can also control what destinations are final in the choice block. For instance, we may not want bad orders continuing through the rest of the route. We would like them to be routed to the badOrders queue and stop there. In this case, we can use the `stop` method in the DSL:

```
from("jms:incomingOrders")
.choice()
    .when(header("CamelFileName").endsWith(".xml"))
        .to("jms:xmlOrders")
    .when(header("CamelFileName").regex("^.*(csv|csl)$"))
        .to("jms:csvOrders")
    .otherwise()
        .to("jms:badOrders").stop()
.end()
.to("jms:continuedProcessing");
```

Now, any orders entering into the otherwise block will only be sent to the badOrders queue—not to the continuedProcessing queue.

Using message filters

Rider Auto Parts now has a new issue—their QA department has expressed the need to be able to send test orders into the live web frontend of the order system. Our current solution would accept these orders as real and send them to the internal systems for processing. We have suggested that QA should be testing on a development clone of the real system, but management has shot down this idea, citing a limited budget. What we need is a solution that will discard these test messages while still operating on the real orders.

The Message Filter, shown in figure 5, provides a nice way of dealing with this kind of problem. Incoming messages only pass through the filter if a certain condition is met. Messages failing the condition will be dropped.

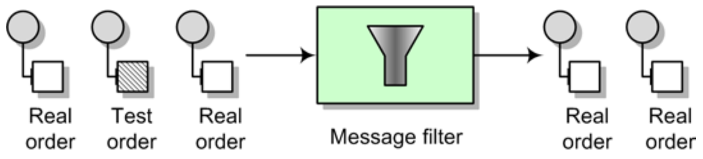


Figure 5 A Message Filter allows you to filter out uninteresting messages based on some condition. In this case, test messages are filtered out.

Let's see how we can implement this using Camel. Recall that the web frontend that Rider Auto Parts uses only sends orders in the XML format, so we can place this filter after the xmlOrders queue where all orders are XML. Test messages have an extra "test" attribute set, so we can use this to do the filtering. A test message looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<order name="motor" amount="1" customer=”foo” test="true"/>
```

The entire solution is implemented in OrderRouterWithFilter.java, which is included with the chapter2/filter project in Camel in Action's source distribution. The filter looks like this:

```
from("jms:xmlOrders").filter().xpath("/order[not(@test)]")
.process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        System.out.println("Received XML order: "
                + exchange.getIn().getHeader("CamelFileName"));
    }
});
```

To run this example, execute the following Maven command on the command line:

```
mvn compile exec:java -Dexec.mainClass=camelinaction.OrderRouterWithFilter
```

This will output the following on the command line:

```
Received XML order: message1.xml
```

We only received one message after the filter because the test message was filtered out.

You may have noticed that we filtered out the test message with an XPath expression. XPath expressions are very useful for creating conditions based on XML payloads. In our case, the expression will evaluate `true` for orders that do not have the "test" attribute.

Another thing worth mentioning about the expression used here is that it was added after the `filter` method rather than as an argument as in the earlier content-based router example. In Camel, an expression that follows a DSL method (a method call on the returned DSL object) is called an *expression clause*. In this case, the DSL method is `filter` and the expression clause is `xpath("/order[not(@test)]")`.

So far, the EIPs we've looked at only sent messages to a single destination. Next we'll take a look at how we can send to multiple destinations.

Using multicast

Often in enterprise applications you will need to send a message to several different destinations for processing. When the list of destinations is known ahead of time and is static, we can add an element to our route that will consume messages from a source endpoint and then send the message out to a list of destinations. Borrowing terminology from computer networking, we call this the multicast EIP.

Currently at Rider Auto Parts, orders are processed in a step by step manner. They are first sent to accounting for validation of customer standing and then to production for manufacture. A bright new manager has suggested that they could improve the speed of operations by sending orders to accounting and production at the same time. This would cut out the delay involved when production waits for the OK from accounting. We have been asked to implement this change to the system.

Using a multicast, we can envision the solution looking like figure 6.

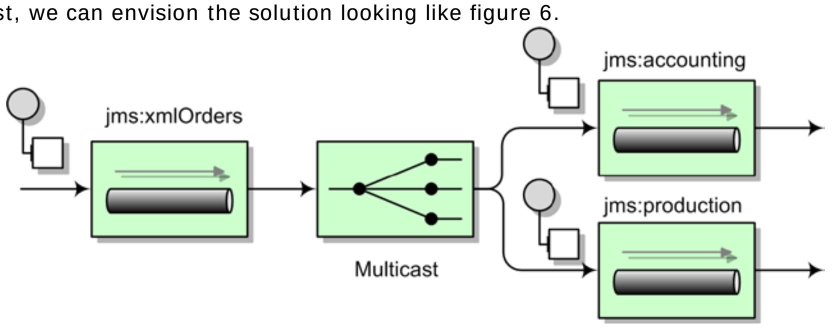


Figure 6 A multicast sends a message to a number of specified recipients

With Camel, we can use the `multicast` method in the Java DSL to implement this solution:

```
from("jms:xmlOrders").multicast().to("jms:accounting", "jms:production");
```

To run this example, go to the chapter2/multicast directory in Camel in Action's source and run this command:

```
mvn compile exec:java
-Dexec.mainClass=camelinaction.OrderRouterWithMulticast
```

You should see the following output on the command line:

```
Accounting received order: message1.xml
Production received order: message1.xml
```

The two lines of output are coming from two test routes that consume from the accounting and production queues and then output text to the console that qualifies the message.

By default, the multicast sends message copies sequentially. In the preceding example, a message is sent to the accounting queue and then to the production queue. But what if we wanted to send them in parallel?

PARALLEL MULTICAST

Sending messages in parallel using the multicast involves only one extra DSL method: `parallelProcessing`. Extending our previous multicast example, we can add the `parallelProcessing` method as follows:

```
from("jms:xmlOrders")
    .multicast().parallelProcessing()
    .to("jms:accounting", "jms:production");
```

This will set up the multicast to distribute messages to the destinations in parallel.

A default thread pool size of 10 is used if you don't specify anything else. If you want to change this default, you can set the underlying java.util.concurrent.ExecutorService that is used to launch new asynchronous sends by using the `executorService` DSL method. Here's an example:

```
ExecutorService executor = Executors.newFixedThreadPool(16);

from("jms:xmlOrders")
    .multicast().parallelProcessing().executorService(executor)
    .to("jms:accounting", "jms:production");
```

Here, we increased the maximum number of threads to 16, in order to handle a larger number of incoming requests.

By default, the multicast will continue sending messages to destinations even if one fails. In your application, though, you may consider the whole process as failed if one destination failed. What do you do in this case?

STOPPING THE MULTICAST ON EXCEPTION

Our multicast solution at Rider Auto Parts suffers from a problem: if the order failed to send to the accounting queue, it may take longer to track down the order from production and bill the customer. To solve this problem, we can take advantage of the `stopOnException` feature of the multicast. When enabled, this feature will stop the multicast on the first exception caught, so we can take necessary action.

To enable this feature, use the `stopOnException` method as follows:

```
from("jms:xmlOrders")
    .multicast().stopOnException()
    .to("jms:accounting", "jms:production");
```

To handle the exception coming back from this route, you will need to use Camel's error handling facilities.

Summary

In this article, we covered one of the core abilities of Camel: routing messages. We showed you how to apply several EIP implementations in Camel and how to use them. With this knowledge, you can create Camel applications that do useful tasks.

Last updated: April 8, 2011