2 Routing with Camel

This chapter covers

- An overview of routing
- Introducing the Rider Auto Parts scenario
- The basics of FTP and JMS endpoints
- Creating routes using the Java DSL
- Configuring routes in XML
- Routing using EIPs

One of the most important features of Camel is routing; without it, Camel would be a library of transport connectors. In this chapter, you'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 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 by which a message is taken from an input queue and, based on a set of conditions, sent to one of several output queues, as shown in figure 2.1. The input and output queues are unaware of the conditions in between them. The conditional logic is decoupled from the message consumer and producer.

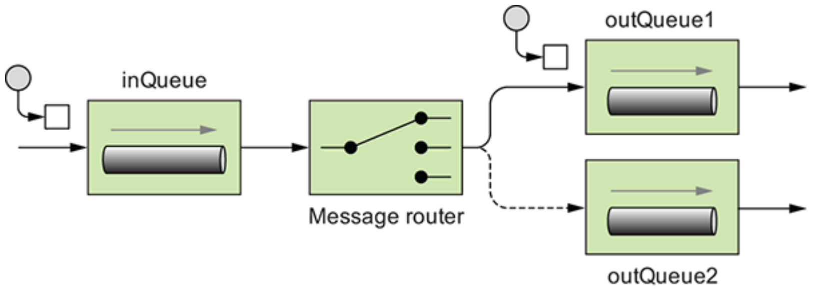


Figure 2.1 A message router consumes messages from an input channel and, depending on a set of conditions, sends the message to one of a set of output channels.

In 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 a system, or even creating the message itself. This message then flows through a processing node, which could be an enterprise integration pattern (EIP), a processor, an interceptor, or another custom creation. The message is finally sent to a target endpoint that's 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.

This chapter introduces the fictional company that we use as the running example throughout the book. To support this company's use case, you'll learn how to communicate over FTP and Java Message Service (JMS) by using Camel's endpoints. Following this, you'll look in depth at the Java-based domain-specific language (DSL) and the XML-based DSL for creating routes. We'll also give you a glimpse of how to design and implement solutions to enterprise integration problems by using EIPs and Camel. By the end of the chapter, you'll be proficient enough to create useful routing applications with Camel.

To start, let's look at the example company used to demonstrate the concepts throughout the book.

2.1 Introducing Rider Auto Parts

Our fictional motorcycle parts business, Rider Auto Parts, supplies parts to motorcycle manufacturers. Over the years, Rider Auto Parts has changed the way it receives orders several times. Initially, orders were placed by uploading comma-separated values (CSV) files to an FTP server. The message format was later changed to XML. Currently, the company provides a website 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, the company 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 2.2.

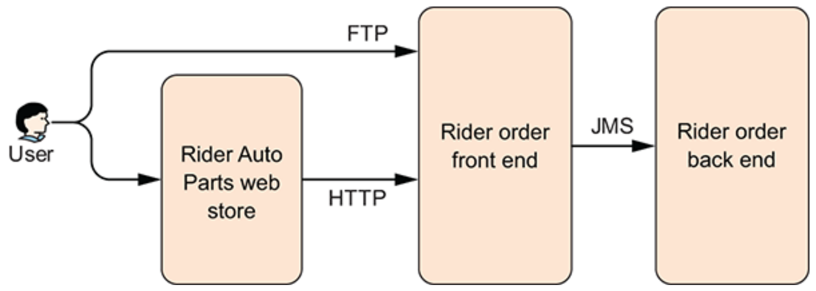


Figure 2.2 A customer has two ways of submitting orders to the Rider Auto Parts order-handling system: either by uploading the raw order file to an FTP server or by submitting an order through the Rider Auto Parts web store. All orders are eventually sent via JMS for processing at Rider Auto Parts.

Rider Auto Parts faces a common problem: over years of operation, it has 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. In this chapter, and throughout the book, you'll help Rider Auto Parts implement its current requirements and new functionality by using Camel.

As a first assignment, you'll need to implement the FTP module in the Rider order front-end system. Later in the chapter, you'll see how back-end services are implemented too. Implementing the FTP module involves the following steps:

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, you need to understand how to communicate over FTP and JMS by using Camel's endpoints. To complete the entire assignment, you need to understand routing with the Java DSL. Let's first take a look at how to use Camel's endpoints.

2.2 Understanding endpoints

As you read in chapter 1, an *endpoint* is an abstraction that models the end of a message channel through which a system can send or receive messages. This section explains how to use URIs to configure Camel to communicate over FTP and JMS. Let's first look at FTP.

2.2.1 Consuming from an FTP endpoint

One of the things that makes Camel easy to use is the endpoint URI. With an endpoint URI, you can identify the component you want to use and the way that component is configured. You can then decide to either send messages to the component configured by this URI, or to consume messages from it.

Take your first Rider Auto Parts assignment, for example. To download new orders from the FTP server, you need to do the following:

1. Connect to the rider.com FTP server on the default FTP port of 21
2. Provide a username of `rider` and password of `secret`
3. Change the directory to `orders`
4. Download any new order files

As shown in figure 2.3, you can easily configure Camel to do this by using URI notation.

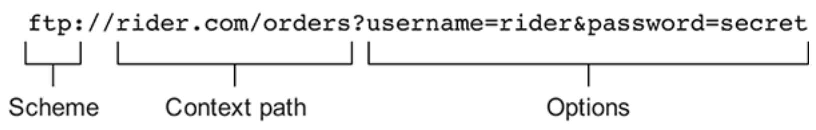


Figure 2.3 A Camel endpoint URI consists of three parts: a scheme, a context path, and a list of options.

Camel first looks up the `ftp` scheme in the component registry, which resolves to `FtpComponent`. `FtpComponent` then works as a factory, creating `FtpEndpoint` based on the remaining context path and options.

The context path of rider.com/orders tells `FtpComponent` that it should log into the FTP server at rider.com on the default FTP port and change the directory to orders. Finally, the only options specified are `username` and `password`, which are used to log in to the FTP server.

TIP For the FTP component, you can also specify the username and password in the context path of the URI, so the following URI is equivalent to the one in figure 2.3: ftp://rider:secret@rider.com/orders. Speaking of passwords, defining them in plain text isn't usually a good idea! You'll find out how to use encrypted passwords in chapter 14.

`FtpComponent` isn't part of the camel-core module, so you have to add a dependency to your project. Using Maven, you add the following dependency to the POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>2.20.1</version>
</dependency>
```

Although this endpoint URI would work equally well in a consumer or producer scenario, you'll be using it to download orders from the FTP server. To do so, you need to use it in a `from` node of Camel's DSL:

```
from("ftp://rider.com/orders?username=rider&password=secret")
```

That's all you need to do to consume files from an FTP server.

The next thing you need to do, as you may recall from figure 2.2, is send the orders you downloaded from the FTP server to a JMS queue. This process requires a little more setup, but it's still easy.

2.2.2 Sending to a JMS endpoint

Camel provides extensive support for connecting to JMS-enabled providers, and we cover all the details in chapter 6. For now, though, we're going to cover just enough so that you can complete your first task for Rider Auto Parts. Recall that you need to download orders from an FTP server and send them to a JMS queue.

What is JMS?

Java Message Service (JMS) is a Java API that allows you to create, send, receive, and read messages. It also mandates that messaging is asynchronous and has specific elements of reliability, such as guaranteed and once-and-only-once delivery. JMS is probably the most widely deployed messaging solution in the Java community.

In JMS, message consumers and producers talk to one another through an intermediary—a JMS destination. As shown in figure 2.4, a destination can be either a queue or a topic. *Queues* are strictly point-to-point; each message has only one consumer. *Topics* operate on a publish/subscribe scheme; a single message may be delivered to many consumers if they've subscribed to the topic.

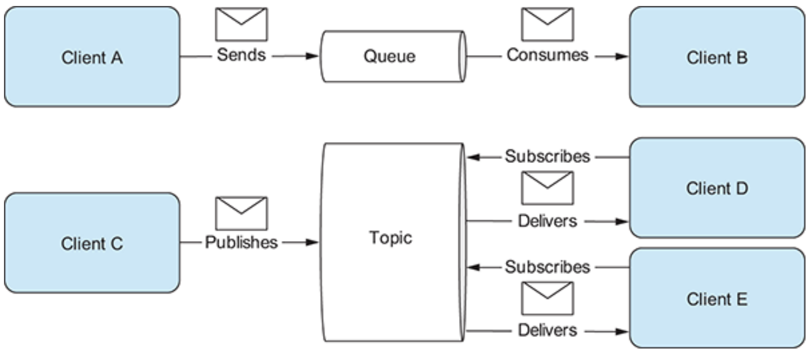


Figure 2.4 There are two types of JMS destinations: queues and topics. The *queue* is a point-to-point channel; each message has only one recipient. A *topic* delivers a copy of the message to all clients that have subscribed to receive it.

JMS also provides a `ConnectionFactory` that clients (for example, Camel) can use to create a connection with a JMS provider. JMS providers are usually referred to as *brokers* because they manage the communication between a message producer and a message consumer.

How to configure Camel to use a JMS provider

To connect Camel to a specific JMS provider, you need to configure Camel's JMS component with an appropriate `ConnectionFactory`.

Apache ActiveMQ is one of the most popular open source JMS providers, and it's the primary JMS broker that the Camel team uses to test the JMS component. As such, we'll be using it to demonstrate JMS concepts within the book. For more information on Apache ActiveMQ, we recommend *ActiveMQ in Action* by Bruce Snyder et al. (Manning, 2011).

In the case of Apache ActiveMQ, you can create an `ActiveMQConnectionFactory` that points to the location of the running ActiveMQ broker:

```
ConnectionFactory connectionFactory =
  new ActiveMQConnectionFactory("vm://localhost");
```

The vm://localhost URI means that you should connect to an embedded broker named *localhost* running inside the current JVM. The `vm` transport connector in ActiveMQ creates a broker on demand if one isn't running already, so it's handy for quickly testing JMS applications; for production scenarios, it's recommended that you connect to a broker that's already running. Furthermore, in production scenarios, we recommend that connection pooling be used when connecting to a JMS broker. See chapter 6 for details on these alternate configurations.

Next, when you create your `CamelContext`, you can add the JMS component as follows:

```
CamelContext context = new DefaultCamelContext();
context.addComponent("jms",
    JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));
```

The JMS component and the ActiveMQ-specific connection factory aren't part of the camel-core module. To use these, you need to add dependencies to your Maven-based project. For the plain JMS component, all you have to add is this:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jms</artifactId>
  <version>2.20.1</version>
</dependency>
```

The connection factory comes directly from ActiveMQ, so you need the following dependency:

```
<dependency>
  <groupId>org.apache.activemq</groupId>
```

```
    <artifactId>activemq-all</artifactId>
    <version>5.15.2</version>
  </dependency>
```

Now that you've configured the JMS component to connect to a JMS broker, it's time to look at how URIs can be used to specify the destination.

Using URIs to specify the destination

After the JMS component is configured, you can start sending and receiving JMS messages at your leisure. Because you're using URIs, this is a real breeze to configure.

Let's say you want to send a JMS message to the queue named `incomingOrders`. The URI in this case would be as follows:

```
jms:queue:incomingOrders
```

This is self-explanatory. The `jms` prefix indicates that you're using the JMS component you configured before. By specifying `queue`, the JMS component knows to send to a queue named `incomingOrders`. You could even omit the queue qualifier, because the default behavior is to send to a queue rather than a topic.

NOTE Some endpoints can have an intimidating list of endpoint URI properties. For instance, the JMS component has more than 80 options, many of which are used only in specific JMS scenarios. Camel always tries to provide built-in defaults that fit most cases, and you can always determine the default values by browsing to the component's page in the online Camel documentation. The JMS component is discussed here: http://camel.apache.org/jms.html.

Using Camel's Java DSL, you can send a message to the `incomingOrders` queue by using the `to` keyword like this:

```
...to("jms:queue:incomingOrders")
```

This can be read as sending `to` the `JMS` `queue` named `incomingOrders`.

Now that you know the basics of communicating over FTP and JMS with Camel, you can get back to the routing theme of this chapter and start routing messages!

2.3 Creating routes in Java

In chapter 1, you saw that `RouteBuilder` can be used to create a route and that each `CamelContext` can contain multiple routes. It may not have been obvious, though, that `RouteBuilder` isn't the final route that `CamelContext` will use at runtime; it's a builder of one or more routes, which are then added to `CamelContext`. This is illustrated in figure 2.5.

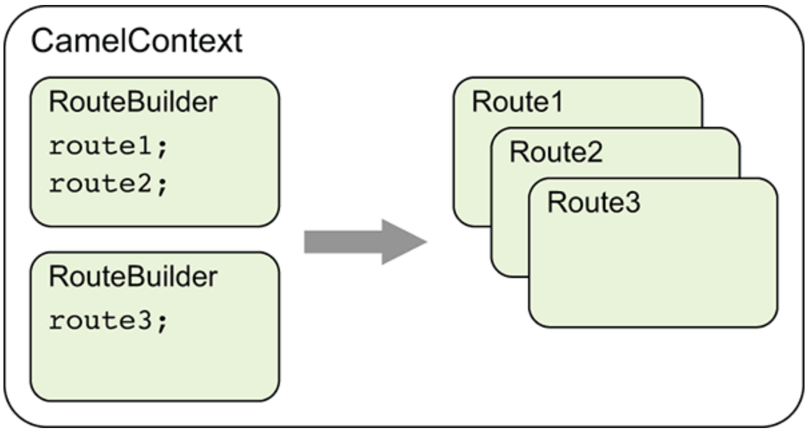


Figure 2.5 `RouteBuilder`s are used to create routes in Camel. Each `RouteBuilder` can create multiple routes.

IMPORTANT This distinction between `RouteBuilder` and routes is an important one. The DSL code you write in `RouteBuilder`, whether that's with the Java or XML DSL, is merely a design-time construct that Camel uses once at startup. So, for instance, the routes that are constructed from `RouteBuilder` are the things that you can debug with your IDE. We cover more about debugging Camel applications in chapter 8.

The `addRoutes` method of `CamelContext` accepts `RoutesBuilder`, not just `RouteBuilder`. The `RoutesBuilder` interface has a single method defined:

```
void addRoutesToCamelContext(CamelContext context) throws Exception;
```

You could in theory use your own custom class to build Camel routes. Not that you'll ever want to do this, though; Camel provides the `RouteBuilder` class for you, which implements `RoutesBuilder`. The `RouteBuilder` class also gives you access to Camel's Java DSL for route creation.

In the next sections, you'll learn how to use `RouteBuilder` and the Java DSL to create simple routes. Then you'll be well prepared to take on the XML DSL in section 2.4 and routing using EIPs in section 2.6.

2.3.1 Using RouteBuilder

The abstract `org.apache.camel.builder.RouteBuilder` class in Camel is one that you'll see frequently. You need to use it anytime you create a route in Java.

To use the `RouteBuilder` class, you extend a class from it and implement the `configure` method, like this:

```
public class MyRouteBuilder extends RouteBuilder {
    public void configure() throws Exception {
        ...
    }
}
```

You then need to add the class to `CamelContext` with the `addRoutes` method:

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new MyRouteBuilder());
```

Alternatively, you can combine the `RouteBuilder` and `CamelContext` configuration by adding an anonymous `RouteBuilder` class directly into `CamelContext`, like this:

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new RouteBuilder() {
    public void configure() throws Exception {
        ...
        }
    });
```

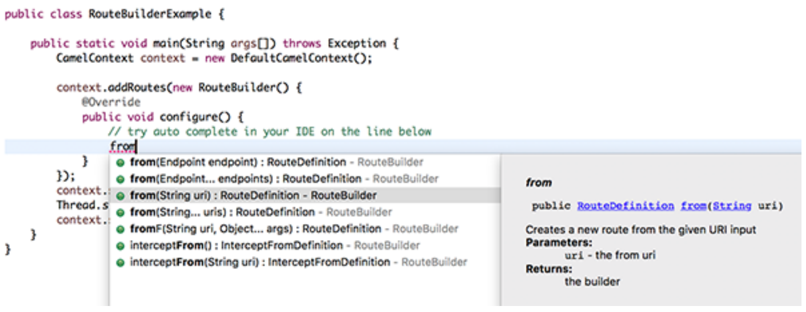
Within the `configure` method, you define your routes by using the Java DSL. We cover the Java DSL in detail in the next section, but you can start a route now to get an idea of how it works.

In chapter 1, you should've downloaded the source code from the book's source code at GitHub and set up Apache Maven. If you didn't do this, please do so now. We'll also be using Eclipse to demonstrate Java DSL concepts.

NOTE Eclipse is a popular open source IDE that you can find at http://eclipse.org. During the book's development, Jon used Eclipse and Claus used IDEA. You can certainly use other Java IDEs as well, or even no IDE, but using an IDE does make Camel development a lot easier. Feel free to skip to the next section if you don't want to see the IDE-related setup. In chapter 19, you will see some additional Camel tooling you can install in Eclipse or IDEA that makes Camel development even better.

After Eclipse is set up, you should import the Maven project in the chapter2/ftp-jms directory of the book's source.

When the ftp-jms project is loaded in Eclipse, open the src/main/java/camelinaction/RouteBuilderExample.java file. As shown in figure 2.6, when you try autocomplete (Ctrl-spacebar in Eclipse) in the `configure` method, you'll be presented with several methods. To start a route, you should use the `from` method.



```
public class RouteBuilderExample {

    public static void main(String args[]) throws Exception {
        CamelContext context = new DefaultCamelContext();

        context.addRoutes(new RouteBuilder() {
            @Override
            public void configure() {
                // try auto complete in your IDE on the line below
                from
            }
        });
        context.
        Thread.s
        context.
    }
}
```

```
from(Endpoint endpoint) : RouteDefinition - RouteBuilder
from(Endpoint... endpoints) : RouteDefinition - RouteBuilder
from(String uri) : RouteDefinition - RouteBuilder
from(String... uris) : RouteDefinition - RouteBuilder
fromF(String uri, Object... args) : RouteDefinition - RouteBuilder
interceptFrom() : InterceptFromDefinition - RouteBuilder
interceptFrom(String uri) : InterceptFromDefinition - RouteBuilder
```

from

```
public RouteDefinition from(String uri)
```

Creates a new route from the given URI input
Parameters:
uri - the from uri
Returns:
the builder

Figure 2.6 Use autocomplete to start your route. All routes start with a `from` method.

The `from` method accepts an endpoint URI as an argument. You can add an FTP endpoint URI to connect to the Rider Auto Parts order server as follows:

```
from("ftp://rider.com/orders?username=rider&password=secret")
```

The `from` method returns a `RouteDefinition` object, on which you can invoke various methods that implement EIPs and other messaging concepts.

Congratulations—you're now using Camel's Java DSL! Let's take a closer look at what's going on here.

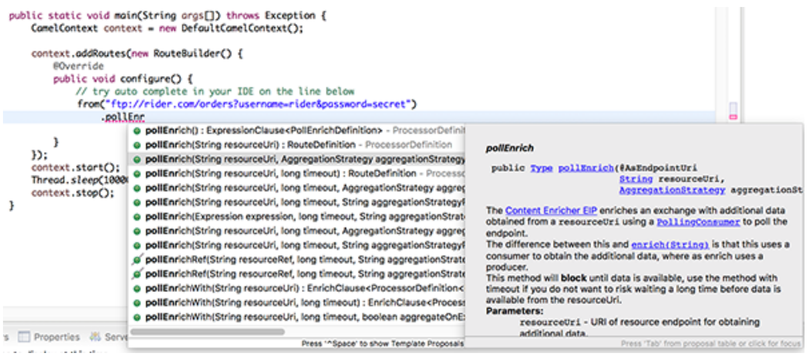
2.3.2 Using the Java DSL

Domain-specific languages (DSLs) are computer languages that target a specific problem domain, rather than a general-purpose domain as most programming languages do. For example, you've probably used the regular expression DSL to match strings of text and found it to be a concise way of matching strings. Doing the same string matching in Java wouldn't be so easy. The regular expression DSL is an *external DSL*; it has a custom syntax and so requires a separate compiler or interpreter to execute. *Internal* DSLs, in contrast, use an existing general-purpose language, such as Java, in such a way that the DSL feels like a language from a particular domain. The most obvious way of doing this is by naming methods and arguments to match concepts from the domain in question.

Another popular way of implementing internal DSLs is by using *fluent interfaces* (a.k.a. *fluent builders*). When using a fluent interface, you build up objects by chaining together method invocations. Methods of this type perform an operation and then return the current object instance.

NOTE For more information on internal DSLs, see Martin Fowler's "Domain Specific Language" entry on his bliki (blog plus wiki) at www.martinfowler.com/bliki/DomainSpecificLanguage.html. He also has an entry on "Fluent Interfaces" at www.martinfowler.com/bliki/FluentInterface.html. For more information on DSLs in general, we recommend *DSLs in Action* by Debasish Ghosh (Manning, 2010).

Camel's domain is enterprise integration, so the Java DSL is a set of fluent builders that contain methods named after terms from the EIP book. In the Eclipse editor, take a look at what's available using autocomplete after a `from` method in the `RouteBuilder`. You should see something like what's shown in figure 2.7. The screenshot shows a couple of EIPs—the Enricher and Recipient List—and there are many others that we'll discuss later.



```
public static void main(String args[]) throws Exception {
    CamelContext context = new DefaultCamelContext();

    context.addRoutes(new RouteBuilder() {
        @Override
        public void configure() {
            // try auto complete in your IDE on the line below
            from("ftp://rider.com/orders?username=rider&password=secret")
                .pollEnr

        }
    });
    context.start();
    Thread.sleep(1000
    context.stop();
}
```

```
pollEnrich() : ExpressionClause<PollEnrichDefinition> - ProcessorDefinit
pollEnrich(String resourceUri) : RouteDefinition - ProcessorDefinition
pollEnrich(String resourceUri, AggregationStrategy aggregationStrategy
pollEnrich(String resourceUri, long timeout) : RouteDefinition - Processo
pollEnrich(String resourceUri, long timeout, AggregationStrategy aggreg
pollEnrich(String resourceUri, long timeout, String aggregationStrategyR
pollEnrich(Expression expression, long timeout, String aggregationStrat
pollEnrich(String resourceUri, long timeout, AggregationStrategy aggreg
pollEnrich(String resourceUri, long timeout, String aggregationStrategyR
pollEnrichRef(String resourceRef, long timeout, String aggregationStrat
pollEnrichRef(String resourceRef, long timeout, String aggregationStrat
pollEnrichWith(String resourceUri) : EnrichClause<ProcessorDefinition<
pollEnrichWith(String resourceUri, long timeout) : EnrichClause<Process
pollEnrichWith(String resourceUri, long timeout, boolean aggregateOnE
```

pollEnrich

```
public Type pollEnrich(@AsEndpointUri
                       String resourceUri,
                       AggregationStrategy aggregationSt
```

The Content Enricher EIP enriches an exchange with additional data obtained from a `resourceUri` using a `PollingConsumer` to poll the endpoint.
The difference between this and `enrich(String)` is that this uses a consumer to obtain the additional data, where as enrich uses a producer.
This method will **block** until data is available, use the method with timeout if you do not want to risk waiting a long time before data is available from the resourceUri.
Parameters:
`resourceUri` - URI of resource endpoint for obtaining additional data.

Properties
Serve
Press '^Space' to show Template Proposals
Press 'Tab' from proposal table or click for focus

Figure 2.7 After the `from` method, use your IDE's autocomplete feature to get a list of EIPs (such as Enricher and Recipient List) and other useful integration functions.

For now, select the `to` method, pass in the string `"jms:incomingOrders",` and finish the route with a semicolon. Each Java statement that starts with a `from` method in the `RouteBuilder` creates a new route. This new route now completes your first task at Rider Auto Parts: consuming orders from an FTP server and sending them to the `incomingOrders` JMS queue. If you want, you can load up the completed example from the book's source code, in chapter2/ftp-jms, and open src/main/java/camelinaction/FtpToJMSExample.java. The code is shown in the following listing.

Listing 2.1 Polling for FTP messages and sending them to the `incomingOrders` queue

```
import javax.jms.ConnectionFactory;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.camel.CamelContext;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.jms.JmsComponent;
import org.apache.camel.impl.DefaultCamelContext;

public class FtpToJMSExample {
    public static void main(String args[]) throws Exception {
        CamelContext context = new DefaultCamelContext();
        ConnectionFactory connectionFactory =
            new ActiveMQConnectionFactory("vm://localhost");
        context.addComponent("jms",
            JmsComponent.jmsComponentAutoAcknowledge(connectionFactory))

        context.addRoutes(new RouteBuilder() {
            public void configure() {
          from("ftp://rider.com/orders"    ①
```

①

Java statement that forms a route

```
            + "?username=rider&password=secret")    ①
            .to("jms:incomingOrders");    ①
            }
        });

        context.start();
        Thread.sleep(10000);
        context.stop();
    }
}
```

NOTE Because you're consuming from ftp://rider.com, which doesn't exist, you can't run this example. It's useful only for demonstrating the Java DSL constructs. For runnable FTP examples, see chapter 6.

As you can see, this listing includes a bit of boilerplate setup and configuration, but the solution to the problem is concisely defined within the `configure` method as a single Java statement ❶. The `from` method tells Camel to consume messages from an FTP endpoint, and the `to` method instructs Camel to send messages to a JMS endpoint.

The flow of messages in this simple route can be viewed as a basic pipeline: the output of the consumer is fed into the producer as input. This is depicted in figure 2.8.

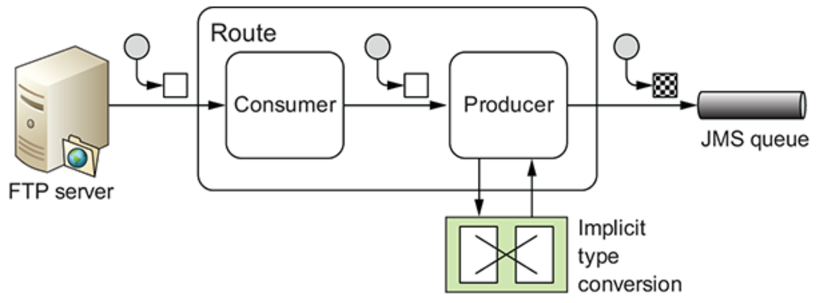


Figure 2.8 The payload conversion from file to JMS message is done automatically.

One thing you may have noticed is that we didn't do any conversion from the FTP file type to the JMS message type—this was done automatically by Camel's type-converter facility. You can force type conversions to occur at any time during a route, but often you don't have to worry about them at all. Data transformation and type conversion is covered in detail in chapter 3.

You may be thinking now that although this route is nice and simple, it'd be nice to see what's going on in the middle of the route. Fortunately, Camel always lets the developer stay in control by providing ways to hook into flows or inject behavior into features. There's a simple way of getting access to the message by using a processor, and we'll discuss that next.

Adding a processor

The `Processor` interface in Camel is an important building block of complex routes. It's a simple interface, having a single method:

```
public void process(Exchange exchange) throws Exception;
```

This gives you full access to the message exchange, letting you do pretty much whatever you want with the payload or headers.

All EIPs in Camel are implemented as processors. You can even add a simple processor to your route inline, like so:

```
from("ftp://rider.com/orders?username=rider&password=secret")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            System.out.println("We just downloaded: "
                + exchange.getIn().getHeader("CamelFileName"));
        }
    })
    .to("jms:incomingOrders");
```

This route will now print the filename of the order that was downloaded before sending it to the JMS queue.

By adding this processor into the middle of the route, you've added it to the conceptual pipeline we mentioned earlier, as illustrated in figure 2.9. The output of the FTP consumer is fed into the processor as input; the processor doesn't modify the message payload or headers, so the exchange moves on to the JMS producer as input.

NOTE Many components, such as `FileComponent` and `FtpComponent`, set useful headers describing the payload on the incoming message. In the previous example, you used the `CamelFileName` header to retrieve the filename of the file that was downloaded via FTP. The component pages of the online documentation contain information about the headers set for each individual component. You'll find information about the FTP component at http://camel.apache.org/ftp.html.

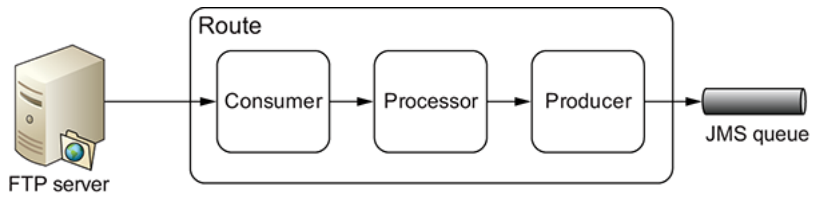


Figure 2.9 With a processor in the mix, the output of the FTP consumer is now fed into the processor, and then the output of the processor is fed into the JMS producer.

Camel's main method for creating routes is through the Java DSL. It is, after all, built into the camel-core module. There are other ways of creating routes, though, some of which may better suit your situation. For instance, Camel provides extensions for writing routes in XML, as we'll discuss next.

2.4 Defining routes in XML

The Java DSL is certainly a more powerful option for the experienced Java developer and can lead to more-concise route definitions. But having the ability to define the same thing in XML opens a lot of possibilities. Maybe some users writing Camel routes aren't the most comfortable with Java; for example, we know many system administrators who handily write up Camel routes to solve integration problems but have never used Java in their lives. The XML configuration also makes nice graphical tooling[1] that has round-trip capabilities possible; you can edit both the XML and graphical representation of a route, and both are kept in sync. Round-trip tooling with Java is possible, but it's a seriously hard thing to do, so none is yet available.

[1.] You can find out more about tooling options for Camel in Chapter 19.

At the time of this writing, you can write XML routes in two Inversion of Control (IoC) Java containers: Spring and OSGi Blueprint. An IoC framework allows you to "wire" beans together to form applications. This wiring is typically done through an XML configuration file. This section gives you a quick introduction to creating applications with Spring so the IoC concept becomes clear. We'll then show you how Camel uses Spring to form a replacement or complementary solution to the Java DSL.

NOTE For a more comprehensive view of Spring, we recommend *Spring in Action* by Craig Walls (Manning, 2014). OSGi Blueprint is covered nicely in *OSGi in Action* by Richard S. Hall et al. (Manning, 2011).

The setup is certainly different between Spring and OSGi Blueprint, yet both have identical route definitions, so we cover only Spring-based examples in this chapter. Throughout the rest of the book, we refer to routes in Spring or Blueprint as just the *XML DSL*.

2.4.1 Bean injection and Spring

Creating an application from beans by using Spring is simple. All you need are a few Java beans (classes), a Spring XML configuration file, and `ApplicationContext`. `ApplicationContext` is similar to `CamelContext`, in that it's the runtime container for Spring. Let's look at a simple example.

Consider an application that prints a greeting followed by your user-name. In this application, you don't want the greeting to be hardcoded, so you can use an interface to break this dependency. Consider the following interface:

```
public interface Greeter {
    public String sayHello();
}
```

This interface is implemented by the following classes:

```
public class EnglishGreeter implements Greeter {
    public String sayHello() {
        return "Hello " + System.getProperty("user.name");
    }
}
public class DanishGreeter implements Greeter {
    public String sayHello() {
        return "Davs " + System.getProperty("user.name");
    }
}
```

You can now create a greeter application as follows:

```
public class GreetMeBean {
    private Greeter greeter;

    public void setGreeter(Greeter greeter) {
        this.greeter = greeter;
    }
    public void execute() {
        System.out.println(greeter.sayHello());
    }
}
```

This application will output a different greeting depending on how you configure it. To configure this application using Spring XML, you could do something like this:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="myGreeter" class="camelinaction.EnglishGreeter"/>
  <bean id="greetMeBean" class="camelinaction.GreetMeBean">
    <property name="greeter" ref="myGreeter"/>
  </bean>
</beans>
```

This XML file instructs Spring to do the following:

1. Create an instance of `EnglishGreeter` and name the bean `myGreeter`
2. Create an instance of `GreetMeBean` and name the bean `greetMeBean`
3. Set the reference of the `greeter` property of the `GreetMeBean` to the bean named `myGreeter`

This configuring of beans is called *wiring*.

To load this XML file into Spring, you can use the `ClassPathXmlApplicationContext`, which is a concrete implementation of `ApplicationContext` that's provided with the Spring framework. This class loads Spring XML files from a location specified on the classpath.

Here's the final version of `GreetMeBean`:

```
public class GreetMeBean {
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");
        GreetMeBean bean = (GreetMeBean) context.getBean("greetMeBean");
        bean.execute();
    }
}
```

The `ClassPathXmlApplicationContext` you instantiate here loads up the bean definitions you saw previously in the beans.xml file. You then call `getBean` on the context to look up the bean with the `greetMeBean` ID in the Spring registry. All beans defined in this file are accessible in this way.

To run this example, go to the chapter2/spring directory in the book's source code and run this Maven command:

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

This will output something like the following on the command line:

```
Hello janstey
```

If you had wired in `DanishGreeter` instead (that is, used the `camelinaction.DanishGreeter` class for the `myGreeter` bean), you'd have seen something like this on the console:

```
Davs janstey
```

This example may seem simple, but it should give you an understanding of what Spring and, more generally, an IoC container, really is. How does Camel fit into this? Camel can be configured as if it were another bean. Recall how you configured the JMS component to connect to an ActiveMQ broker in section 2.2.2 by using Java code:

```
ConnectionFactory connectionFactory =
    new ActiveMQConnectionFactory("vm://localhost");
CamelContext context = new DefaultCamelContext();
context.addComponent("jms",
    JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));
```

You could have done this in Spring by using the bean terminology, as follows:

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://localhost"/>
    </bean>
  </property>
</bean>
```

In this case, if you send to an endpoint such as `"jms:incomingOrders"`, Camel will look up the `jms` bean, and if it's of type `org.apache.camel.Component`, it will use that. So you don't have to manually add components to `CamelContext`—a task that you did manually in section 2.2.2 for the Java DSL.

But where's `CamelContext` defined in Spring? Well, to make things easier on the eyes, Camel uses Spring extension mechanisms to provide custom XML syntax for Camel concepts within the Spring XML file. To load up `CamelContext` in Spring, you can do the following:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd">
  ...
  <camelContext xmlns="http://camel.apache.org/schema/spring"/>
</beans>
```

This automatically starts `SpringCamelContext`, which is a subclass of `DefaultCamelContext`, which you used for the Java DSL. Also notice that you have to include the `http://camel.apache.org/schema/spring/camel-spring.xsd` XML schema definition in the XML file; this is needed to import the custom XML elements.

This snippet alone isn't going to do much for you. You need to tell Camel what routes to use, as you did when using the Java DSL. The following code uses Spring XML to produce the same results as the code in listing 2.1.

Listing 2.2 A Spring configuration that produces the same results as listing 2.1

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd">

  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="vm://localhost" />
      </bean>
    </property>
  </bean>
  <bean id="ftpToJmsRoute" class="camelinaction.FtpToJMSRoute"/>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <routeBuilder ref="ftpToJmsRoute"/>
  </camelContext>
</beans>
```

You may have noticed that we're referring to the `camelinaction.FtpToJMSRoute` class as a `RouteBuilder`. To reproduce the Java DSL example in listing 2.1, you have to factor out the anonymous `RouteBuilder` into its own named class. The `FtpToJMSRoute` class looks like this:

```
public class FtpToJMSRoute extends RouteBuilder {
  public void configure() {
    from("ftp://rider.com/orders?username=rider&password=secret")
      .to("jms:incomingOrders");
  }
}
```

Now that you know the basics of Spring and how to load Camel inside it, we can go further by looking at how to write Camel routing rules purely in XML—no Java DSL required.

2.4.2 The XML DSL

What we've seen of Camel's integration with Spring is adequate, but it isn't taking full advantage of Spring's methodology of configuring applications using no code. To completely invert the control of creating applications using Spring XML, Camel provides custom XML extensions that we call the *XML DSL*. The XML DSL allows you to do almost everything you can do in the Java DSL.

Let's continue with the Rider Auto Parts example shown in listing 2.2, but this time you'll specify the routing rules defined in `RouteBuilder` purely in XML. The Spring XML in the following listing does this.

Listing 2.3 An XML DSL example that produces the same results as listing 2.1

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd">

  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="vm://localhost" />
      </bean>
    </property>
  </bean>
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="ftp://rider.com/orders?username=rider&amp;password=secr
      <to uri="jms:incomingOrders"/>
    </route>
  </camelContext>
</beans>
```

In this listing, under the `camelContext` element, you replace `route-Builder` with the `route` element. Within the `route` element, you specify the route by using elements with names similar to ones used inside the Java DSL `RouteBuilder`. Notice that we had to modify the FTP endpoint URI to ensure that it's valid XML. The ampersand character (`&`) used to define extra URI options is a reserved character in XML, so you have to escape it by using `&amp;`. With this small change, this listing is functionally equivalent to the Java DSL version in listing 2.1 and the Spring plus Java DSL combo in listing 2.2.

In the book's source code, we changed the `from` method to consume messages from a local file directory instead. The new route looks like this:

```
<route>
  <from uri="file:src/data?noop=true"/>
  <to uri="jms:incomingOrders"/>
</route>
```

The file endpoint will load order files from the relative src/data directory. The `noop` property configures the endpoint to leave the file as is after processing; this option is useful for testing. In chapter 6, you'll see how Camel allows you to delete or move the files after processing.

This route won't display anything interesting yet. You need to add a processing step for testing.

Adding a processor

Adding processing steps is simple, as in the Java DSL. Here you'll add a custom processor as you did in section 2.3.2.

Because you can't refer to an anonymous class in Spring XML, you need to factor out the anonymous processor into the following class:

```
public class DownloadLogger implements Processor {
    public void process(Exchange exchange) throws Exception {
        System.out.println("We just downloaded: "
                + exchange.getIn().getHeader("CamelFileName"));
    }
}
```

You can now use the processor in your XML DSL route as follows:

```
<bean id="downloadLogger" class="camelinaction.DownloadLogger"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file:src/data?noop=true"/>
    <process ref="downloadLogger"/>
    <to uri="jms:incomingOrders"/>
  </route>
</camelContext>
```

Now you're ready to run the example. Go to the chapter2/spring directory in the book's source code and run this Maven command:

```
mvn clean compile camel:run
```

Because there's only one message file named message1.xml in the src/data directory, this outputs something like the following on the command line:

```
We just downloaded: message1.xml
```

What if you wanted to print this message after consuming it from the `incomingOrders` queue? To do this, you need to create another route.

Using multiple routes

You may recall that in the Java DSL each Java statement starting with a `from` creates a new route. You can also create multiple routes with the XML DSL. To do this, add a `route` element within the `camelContext` element.

For example, move the `DownloadLogger` processor into a second route, after the order gets sent to the `incomingOrders` queue:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file:src/data?noop=true"/>
    <to uri="jms:incomingOrders"/>
  </route>
  <route>
    <from uri="jms:incomingOrders"/>
    <process ref="downloadLogger"/>
  </route>
</camelContext>
```

Now you're consuming the message from the `incomingOrders` queue in the second route, so the downloaded message will be printed after the order is sent via the queue.

Choosing which DSL to use

Which DSL is best to use in a particular scenario is a common question for Camel users, but it mostly comes down to personal preference. If you like working with Spring or like defining things in XML, you may prefer a pure XML approach. If you want to be hands-on with Java, maybe a pure Java DSL approach is better for you.

In either case, you'll be able to access nearly all of Camel's functionality. The Java DSL is a slightly richer language to work with because you have the full power of the Java language at your fingertips. Also, some Java DSL features, such as value builders (for building expressions and predicates), aren't available in the XML DSL. On the other hand, using Spring XML gives you access to the wonderful object construction capabilities as well as commonly used Spring abstractions for things like database connections and JMS integration. The XML DSL also makes nice graphical tooling possible that has round-trip capabilities: you can edit both the XML and graphical representation of a route, and both are kept in sync.[2] A common compromise is to use both Spring XML and the Java DSL, which is one of the topics we'll cover next.

[2.] See Bilgin Ibryam's blog post on which Camel DSL to use: http://www.ofbizian.com/2017/12/which-camel-dsl-to-choose-and-why.html.

2.4.3 Using Camel and Spring

Whether you write your routes in the Java or XML DSL, running Camel in a Spring container gives you many other benefits. For one, if you're using the XML DSL, you don't have to recompile any code when you want to change your routing rules. Also, you gain access to Spring's portfolio of database connectors, transaction support, and more.

Let's take a closer look at what other Spring integrations Camel provides.

Finding route builders

Using the Spring `CamelContext` as a runtime and the Java DSL for route development is a great way to use Camel. You saw before in listing 2.2 that you can explicitly tell the Spring `CamelContext` what route builders to load. You can do this by using the `routeBuilder` element:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <routeBuilder ref="ftpToJmsRoute"/>
</camelContext>
```

Being this explicit results in a clean and concise definition of what is being loaded into Camel.

Sometimes, though, you may need to be a bit more dynamic. This is where the `packageScan` and `contextScan` elements come in:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <packageScan>
    <package>camelinaction.routes</package>
  </packageScan>
</camelContext>
```

This `packageScan` element will load all `RouteBuilder` classes found in the `camelinaction.routes` package, including all subpackages.

You can even be a bit pickier about what route builders are included:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <packageScan>
    <package>camelinaction.routes</package>
    <excludes>**.*Test*</excludes>
    <includes>**.*</includes>
  </packageScan>
</camelContext>
```

In this case, you're loading all route builders in the `camelinaction.routes` package, except for ones with `Test` in the class name. The matching syntax is similar to what's used in Apache Ant's file pattern matchers.

The `contextScan` element takes advantage of Spring's component-scan feature to load any Camel route builders that are marked with the `org.springframework.stereotype.@Component` annotation. Let's modify the `FtpToJMSRoute` class to use this annotation:

```
@Component
public class FtpToJMSRoute extends RouteBuilder {
    public void configure() {
    from("ftp://rider.com" +
        "/orders?username=rider&password=secret")
        .to("jms:incomingOrders");
    }
}
```

You can now enable the component scanning by using the following configuration in your Spring XML file:

```
<context:component-scan base-package="camelinaction.routes"/>
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <contextScan/>
</camelContext>
```

This will load up any Camel route builders within the `camelinaction.routes` package that have the `@Component` annotation.

Under the hood, some of Camel's components, such as the JMS component, are built on top of abstraction libraries from Spring. This often explains why configuring those components is easy in Spring.

CONFIGURING COMPONENTS AND ENDPOINTS

You saw in section 2.4.1 that components could be defined in Spring XML and would be picked up automatically by Camel. For instance, look at the JMS component again:

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://localhost"/>
    </bean>
  </property>
</bean>
```

The bean `id` defines what this component will be called. This gives you the flexibility to give the component a more meaningful name based on the use case. Your application may require the integration of two JMS brokers, for instance. One could be for Apache ActiveMQ and another could be for WebSphere MQ:

```
<bean id="activemq" class="org.apache.camel.component.jms.JmsComponent">
  ...
</bean>
<bean id="wmq" class="org.apache.camel.component.jms.JmsComponent">
  ...
</bean>
```

You could then use URIs such as `activemq:myActiveMQQueue` or `wmq:myWebSphereQueue`. Endpoints can also be defined by using Camel's Spring XML extensions. For example, you can break out the FTP endpoint for connecting to the Rider Auto Parts legacy order server into an `<endpoint>` element that's highlighted in bold here:

```
    <camelContext xmlns="http://camel.apache.org/schema/spring">
      <endpoint id="ridersFtp"
        uri="ftp://rider.com/orders?username=rider&amp;password=secret"/
      <route>
        <from ref="ridersFtp"/>
        <to uri="jms:incomingOrders"/>
      </route>
    </camelContext>
```

NOTE You may notice that credentials have been added directly into the endpoint URI, which isn't always the best solution. A better way is to refer to credentials that are defined and sufficiently protected elsewhere. In section 14.1 of chapter 14, you can see how the Camel Properties component or Spring property placeholders are used to do this.

For longer endpoint URIs, it's often easier to read them if you break them up over several lines. See the previous route with the endpoint URI options broken into separate lines:

```
      <camelContext xmlns="http://camel.apache.org/schema/spring">
        <endpoint id="ridersFtp" uri="ftp://rider.com/orders?
                                    username=rider&amp;
                                    password=secret"/>
        <route>
          <from ref="ridersFtp"/>
          <to uri="jms:incomingOrders"/>
        </route>
      </camelContext>
```

Importing configuration and routes

A common practice in Spring development is to separate an application's wiring into several XML files. This is mainly done to make the XML more readable; you probably wouldn't want to wade through thousands of lines of XML in a single file without some separation.

Another reason to separate an application into several XML files is the potential for reuse. For instance, another application may require a similar JMS setup, so you can define a second Spring XML file called jms-setup.xml with these contents:

```
   <beans xmlns="http://www.springframework.org/schema/beans"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="
```

```
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
      <property name="connectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
          <property name="brokerURL" value="vm://localhost" />
        </bean>
      </property>
    </bean>
  </beans>
```

This file could then be imported into the XML file containing `CamelContext` by using the following line:

```
<import resource="jms-setup.xml"/>
```

Now `CamelContext` can use the JMS component configuration even though it's defined in a separate file.

Other useful things to define in separate files are the XML DSL routes themselves. Because route elements need to be defined within a `camel-Context` element, an additional concept is introduced to define routes. You can define routes within a `routeContext` element, as shown here:

```
<routeContext id="ftpToJms" xmlns="http://camel.apache.org/schema/spring
  <route>
    <from uri="ftp://rider.com/orders?username=rider&password=secret"/>
    <to uri="jms:incomingOrders"/>
  </route>
</routeContext>
```

This `routeContext` element could be in another file or in the same file. You can then import the routes defined in this `routeContext` with the `routeContextRef` element. You use the `routeContextRef` element inside `camelContext` as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <routeContextRef ref="ftpToJms"/>
</camelContext>
```

If you import `routeContext` into multiple `CamelContext`s, a new instance of the route is created in each. In the preceding case, two identical routes, with the same endpoint URIs, will lead to them competing for the same resource. In this case, only one route at a time will receive a particular file from FTP. In general, you should take care when reusing routes in multiple `CamelContext`s.

Setting advanced Spring configuration options

Many other configuration options are available when using the Spring `CamelContext`:

- Pluggable bean registries are discussed in chapter 4.
- The configuration of interceptors is covered in chapter 9.
- Stream caching, fault handling and startup are mentioned in chapter 15.
- The Tracer mechanism is covered in chapter 16.

2.5 Endpoints revisited

In section 2.2, we covered the basics of endpoints in Camel. Now that you've seen both Java and XML routes in action, it's time to introduce more-advanced endpoint configurations.

2.5.1 Sending to dynamic endpoints

Endpoint URIs like the JMS one shown in section 2.2.2 are evaluated just once when Camel starts up, so they're static entities in Camel. This is fine for most scenarios, when you know ahead of time what the destination names will be called. But what if you need to determine these names at runtime? Static endpoint URIs provided to the `to` method will be of no use in this case, because they're evaluated only once at startup. Camel provides an additional DSL method for this: `toD`.

For example, say you want to make the endpoint URI point to a destination name stored as a message header. The following does that:

```
.toD("jms:queue:${header.myDest}");
```

And in the XML DSL:

```
<toD uri="jms:queue:${header.myDest}"/>
```

This endpoint URI uses a Simple expression within the `${ }` placeholders to return the value of the `myDest` header in the incoming message. If `my-Dest` is `incomingOrders`, the resulting endpoint URI will be `jms:queue:incomingOrders`, as we had before in the static case. If you were wondering about the Simple language, it's a lightweight expression language built into Camel's core. We go over Simple in more detail in section 2.6.1 of this chapter, and appendix A provides a complete reference.

To run this example yourself, go to the chapter2/ftp-jms directory in the book's source code and run this Maven command:

```
mvn test -Dtest=FtpToJMSWithDynamicToTest
```

2.5.2 Using property placeholders in endpoint URIs

Rather than having hardcoded endpoint URIs, Camel allows you to use property placeholders in the URIs to replace the dynamic parts. By *dynamic*, we mean that values will be replaced when Camel starts up, not on every new message, as in the case of `toD` described in the previous section.

One common usage of property placeholders is in testing. A Camel route is often tested in different environments—you may want to test it locally on your laptop, and then later on a dedicated test platform, and so forth. But you don't want to rewrite tests every time you move to a new environment. That's why you externalize dynamic parts rather than hardcoding them.

Using the Properties component

Camel has a Properties component to support externalizing properties defined in the routes (and elsewhere). The Properties component works in much the same way as Spring property placeholders, but it has a few noteworthy improvements:

- It's built in the camel-core JAR, which means it can be used without the need for Spring or any third-party framework.

- It can be used in all the DSLs, such as the Java DSL, and isn't limited to Spring XML files.
- It supports masking sensitive information by plugging in third-party encryption libraries.

For more details on the Properties component, see the Camel documentation: http://camel.apache.org/properties.html.

TIP You can use the Jasypt component to encrypt sensitive information in the properties file. For example, you may not want to have passwords in clear text in the properties file. You can read more about the Jasypt component in chapter 14.

To ensure that the property placeholder is loaded and in use as early as possible, you have to configure `PropertiesComponent` when `CamelContext` is created:

```
CamelContext context = new DefaultCamelContext();
PropertiesComponent prop = camelContext.getComponent(
    "properties", PropertiesComponent.class);
prop.setLocation("classpath:rider-test.properties");
```

In the rider-test.properties file, you define the externalized properties as key-value pairs:

```
myDest=incomingOrders
```

`RouteBuilder`s can then take advantage of the externalized properties directly in the endpoint URI, as shown in bold in this route:

```
return new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("file:src/data?noop=true")
            .to("jms:{{myDest}}");
        from("jms:incomingOrders")
            .to("mock:incomingOrders");
    }
};
```

You should notice that the Camel syntax for property placeholders is a bit different than for Spring property placeholders. The Camel Properties component uses the `{{key}}` syntax, whereas Spring uses `${key}`.

You can try this example by using the following Maven goal from the chapter2/ftp-jms directory:

```
mvn test -Dtest=FtpToJMSWithPropertyPlaceholderTest
```

Setting this up in XML is a bit different, as you'll see in the next section.

Using property placeholders in the XML DSL

To use the Camel Properties component in Spring XML, you have to declare it as a Spring bean with the ID `properties`, as shown here:

```
<bean id="properties"
       class="org.apache.camel.component.properties.PropertiesComponent"
    <property name="location" value="classpath:rider-test.properties"/>
</bean>
```

In the rider-test.properties file, you define the externalized properties as key-value pairs:

```
myDest=incomingOrders
```

The `camelContext` element can then take advantage of the externalized properties directly in the endpoint URI, as shown in bold in this route:

```
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="file:src/data?noop=true"/>
      <to uri="jms:{{myDest}}" />
    </route>
    <route>
      <from uri="jms:incomingOrders"/>
      <to uri="mock:incomingOrders"/>
    </route>
  </camelContext>
```

Instead of using a Spring bean to define the Camel Properties component, you can also use a specialized `<propertyPlaceholder>` within `camel-Context`, as follows:

```
    <camelContext xmlns="http://camel.apache.org/schema/spring">
      <propertyPlaceholder id="properties"
              location="classpath:rider-test.properties"/>
      <route>
        <from uri="file:src/data?noop=true"/>
        <to uri="jms:{{myDest}}"/>
      </route>
      <route>
        <from uri="jms:incomingOrders"/>
        <to uri="mock:incomingOrders"/>
      </route>
    </camelContext>
```

This example is included in the book's source code in the chapter2/spring directory. You can try it by using the following Maven goal:

```
mvn test -Dtest=SpringFtpToJMSWithPropertyPlaceholderTest
```

We'll now cover the same example, but using Spring property placeholders instead of the Camel Properties component.

Using Spring property placeholders

The Spring Framework supports externalizing properties defined in the Spring XML files by using a feature known as Spring *property placeholders*. We'll review the example from the previous section, using Spring property placeholders instead of the Camel Properties component.

The first thing you need to do is set up the route having the endpoint URIs externalized. This could be done as follows. Notice that Spring uses the `${key}` syntax:

```
    <context:property-placeholder properties-ref="properties"/>
    <util:properties id="properties"
```

```
                      location="classpath:rider-test.properties"/>

    <camelContext xmlns="http://camel.apache.org/schema/spring">

      <endpoint id="myDest" uri="jms:${myDest}"/>

      <route>
        <from uri="file:src/data?noop=true"/>
        <to uri="jms:${myDest}"/>
      </route>

      <route>
        <from uri="jms:incomingOrders"/>
        <to uri="mock:incomingOrders"/>
      </route>
    </camelContext>
```

Unfortunately, the Spring Framework doesn't support using placeholders directly in endpoint URIs in the route, so you must define endpoints that include those placeholders by using the `<endpoint>` tag. The following code snippet shows how this is done:

```
    <context:property-placeholder properties-ref="properties"/>
    <util:properties id="properties"    1                      location="classpath
```

1

Loads properties from external file

```
    <camelContext xmlns="http://camel.apache.org/schema/spring">

    <endpoint id="myDest" uri="jms:${myDest}"/>    2
```

2

Defines endpoint using Spring property placeholders

```
      <route>
```

```
        <from uri="file:src/data?noop=true"/>
    <to ref="myDest"/>    ❸
```

---

❸

Refers to endpoint in route

```
      </route>

      <route>
        <from uri="jms:incomingOrders"/>
        <to uri="mock:incomingOrders"/>
      </route>
    </camelContext>
```

To use Spring property placeholders, you must declare the `<context:property-placeholder>` tag where you refer to a `properties` bean ❶ that will load the properties file from the classpath.

In the `camelContext` element, you define an endpoint ❷ that uses a placeholder for a dynamic JMS destination name. The `${myDest}` is a Spring property placeholder that refers to a property with the key `my-Dest`.

In the route, you must refer to the endpoint ❸ instead of using the regular URI notations. Notice the use of the `ref` attribute in the `<to>` tag.

The rider-test.properties properties file contains the following line:

```
myDest=incomingOrders
```

This example is included in the book's source code in the chapter2/spring directory. You can try it by using the following Maven goal:

```
mvn test -Dtest=SpringFtpToJMSWithSpringPropertyPlaceholderTest
```

THE CAMEL PROPERTIES COMPONENT VS. SPRING PROPERTY PLACEHOLDERS

The Camel Properties component is more powerful than the Spring property placeholder mechanism. The latter works only when defining routes using Spring XML, and you have to declare the endpoints in dedicated `<endpoint>` tags for the property placeholders to work.

The Camel Properties component is provided out of the box, which means you can use it without using Spring at all. And it supports the various DSL languages you can use to define routes, such as Java, Spring XML, and Blueprint OSGi XML. On top of that, you can declare the placeholders anywhere in the route definitions.

2.5.3 Using raw values in endpoint URIs

Sometimes values you want to use in a URI will make the URI itself invalid. Take, for example, an FTP password of `++%%w?rd`. Adding this as is would break any URI because it uses reserved characters. You could encode these reserved characters, but that would make things less readable (not that you'd want your password more readable—it's just an example!). Camel's solution is to allow "raw" values in endpoint URIs that don't count toward URI validation. For example, let's use this raw password to connect to the rider.com FTP server:

```
from("ftp://rider.com/orders?username=rider&password=RAW(++%%w?rd)")
```

As you can see, the password is surrounded by `RAW()`, which will make Camel treat this value as a raw value.

2.5.4 Referencing registry beans in endpoint URIs

You've heard the Camel registry mentioned a few times now but haven't seen it in use. We don't dive into detail about the registry here (that's covered in chapter 4), but we'll show a common syntax in endpoint URIs related to the registry. Anytime a Camel endpoint requires an object instance as an option value, you can refer to one in the registry by using the `#` syntax. For example, say you want to fetch only CSV order files from the FTP site. You could define a filter like so:

```
public class OrderFileFilter<T> implements GenericFileFilter<T> {
     public boolean accept(GenericFile<T> file) {
        return file.getFileName().endsWith("csv");
    }
}
```

Add it to the registry:

```
registry.bind("myFilter", new OrderFileFilter<Object>());
```

Then you use the `#` syntax to refer to the named instance in the registry:

```
from("ftp://rider.com/orders?username=rider&password=secret&filter=#myFi
```

With these endpoint configuration techniques behind you, you're ready to tackle more-advanced routing topics by using Camel's implementation of the EIPs.

2.6 Routing and EIPs

So far, we haven't touched much on the EIPs that Camel was built to implement. That's intentional. We want to make sure you have a good understanding of what Camel is doing in the simplest cases before moving on to more-complex examples.

As far as EIPs go, we'll be looking at the Content-Based Router, Message Filter, Multicast, Recipient List, and Wire Tap right away. Other patterns are introduced throughout the book, and chapter 5 covers the most complex EIPs. The complete list of EIPs supported by Camel is available from the Camel website (http://camel.apache.org/eip.html).

For now, let's start by looking at the most well-known EIP: the Content-Based Router.

2.6.1 Using a content-based router

As the name indicates, 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, or part of the payload itself—pretty much anything in the message exchange.

To demonstrate, let's go back to Rider Auto Parts. Some customers have started uploading orders to the FTP server in the newer XML format rather than CSV. You have two types of messages coming in to the `incomingOrders` queue. We didn't touch on this before, but you need to convert the incoming orders into an internal POJO format. You need to do different conversions for the different types of incoming orders.

As a possible solution, you could use the filename extension to determine whether a particular order message should be sent to a queue for CSV orders or a queue for XML orders. This is depicted in figure 2.10.

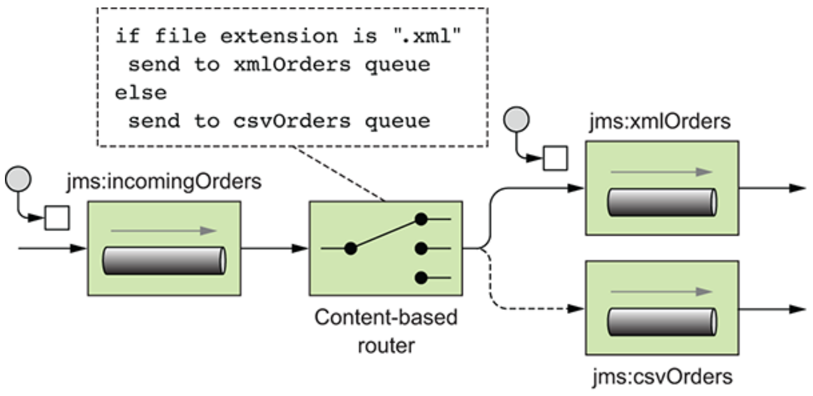


Figure 2.10 The CBR routes messages based on their content. In this case, the filename extension (as a message header) is used to determine which queue to route to.

As you saw earlier, you can use the `CamelFileName` header set by the FTP consumer to get the filename.

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

Camel's creators could have chosen `contentBasedRouter` for the method name, to match the EIP, but they 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 has only 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. You can choose from many expression languages in Camel, some of which include Simple, SpEL, JXPath, MVEL, OGNL, JavaScript, Groovy, XPath, and XQuery. As you'll see in chapter 4, you can even use a method call to a bean as an expression in Camel. In this case, you'll be using the expression builder methods that are part of the Java DSL.

Within `RouteBuilder`, you can start by using the `header` method, which returns an expression that will evaluate to the header value. For example, `header("CamelFileName")` creates an expression that will resolve to the value of the `CamelFileName` header on the incoming exchange. On this expression, you can invoke methods to create a predicate. To check whether the filename extension is equal to .xml, you can use the follow-ing predicate:

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

The completed CBR is shown in the following listing.

Listing 2.4 A complete content-based router using the Java DSL

```
return new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        // load file orders from src/data into the JMS queue
        from("file:src/data?noop=true").to("jms:incomingOrders");

    from("jms:incomingOrders") 1
```

1

Content-based router

```
        .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") 2
```

2

Test routes that print message content

```
        .log("Received XML order: ${header.CamelFileName}") 2
        .to("mock:xml"); 2

    from("jms:csvOrders") 2
        .log("Received CSV order: ${header.CamelFileName}") 2
        .to("mock:csv"); 2
    }
};
```

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

```
mvn test -Dtest=OrderRouterTest
```

This consumes two order files in the chapter2/cbr/src/data directory and outputs 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 ❷. These routes consume messages from the `xmlOrders` and `csvOrders` queues and then print messages by using the `log` DSL method.

THE SIMPLE LANGUAGE

You may have noticed that the string passed into the `log` method has something that looks like properties defined for expansion. This `${header.CamelFileName}` term is from the Simple language, which is Camel's own expression language included with the camel-core module. The Simple language contains many useful variables, functions, and operators that operate on the incoming exchange. A dynamic expression in the Simple language is enclosed with the `${ }` placeholders, as you saw in listing 2.4. Let's consider this example:

```
${header.CamelFileName}
```

Here, `header` maps to the headers of the in message of the exchange. After the dot, you can append any header name that you want to access. At runtime, Camel will return the value of the `CamelFileName` header in the in message. You could also replace your content-based router conditions in listing 2.4 with simple expressions:

```
from("jms:incomingOrders")
    .choice()
        .when(simple("${header.CamelFileName} ends with 'xml'"))
            .to("jms:xmlOrders")
        .when(simple("${header.CamelFileName} ends with 'csv'"))
            .to("jms:csvOrders");
```

Here you use the `ends with` operator to check the end of the string returned from the `${header.CamelFileName}` dynamic simple expression. The Simple language is so useful for Camel applications that we devote appendix A to cover it fully.

You use these routes to test that the router ❶ is working as expected. Route-testing techniques, like use of the Mock component, are discussed in chapter 9.

You can also form an equivalent CBR by using the XML DSL, as shown in listing 2.5. Other than being in XML rather than Java, the main difference is that you use a Simple expression instead of the Java-based predicate ❶. The Simple expression language is a great option for replacing predicates from the Java DSL.

Listing 2.5 A complete content-based router using the XML DSL

```
<route>
 <from uri="file:src/data?noop=true"/>
 <to uri="jms:incomingOrders"/>
</route>

<route>
  <from uri="jms:incomingOrders"/>
  <choice>
    <when>
      <simple>${header.CamelFileName} ends with 'xml'</simple>    ❶
```

❶

Simple expression used instead of Java-based predicate

```
      <to uri="jms:xmlOrders"/>
    </when>
    <when>
      <simple>${header.CamelFileName} ends with 'csv'</simple>    ❶
      <to uri="jms:csvOrders"/>
    </when>
  </choice>
</route>

<route>    ❷
```

Test routes that print message content

```
  <from uri="jms:xmlOrders"/>    ❷
  <log message="Received XML order: ${header.CamelFileName}"/>    ❷
  <to uri="mock:xml"/>    ❷
</route>    ❷

<route>    ❷
  <from uri="jms:csvOrders"/>    ❷
   <log message="Received CSV order: ${header.CamelFileName}"/>    ❷
  <to uri="mock:csv"/>    ❷
</route>    ❷
```

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

```
mvn test -Dtest=SpringOrderRouterTest
```

You'll see output similar to that of the Java DSL example.

Using the otherwise clause

A Rider Auto Parts customer sends CSV orders with the .csl extension. Your current route handles only .csv and .xml files and will drop all orders with other extensions. This isn't a good solution, so you 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 `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. You should be handling bad orders that come in so someone can fix the problem. For this, you 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.

The equivalent route in XML DSL is as follows:

```
<route>
  <from uri="jms:incomingOrders"/>
  <choice>
    <when>
      <simple>${header.CamelFileName} ends with '.xml'</simple>
      <to uri="jms:xmlOrders"/>
    </when>
    <when>
      <simple>${header.CamelFileName} regex '^.*(csv|csl)$'</simple>
      <to uri="jms:csvOrders"/>
    </when>
    <otherwise>
      <to uri="jms:badOrders"/>
    </otherwise>
  </choice>
</route>
```

To run this example, go to the chapter2/cbr directory in the book's source and run one or both of these Maven commands:

```
mvn test -Dtest=OrderRouterOtherwiseTest
mvn test -Dtest=SpringOrderRouterOtherwiseTest
```

This consumes four order files in the chapter2/cbr/src/data_full directory and outputs the following:

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

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

Routing after a content-based router

The CBR may seem like it's 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 you can continue routing after a CBR. One is by using another route, as you did in listing 2.4 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, the `choice` has been closed and another `to` has been added to the route. After each destination with the `choice`, the message will be routed to the `continuedProcessing` queue as well. This is illustrated in figure 2.11.

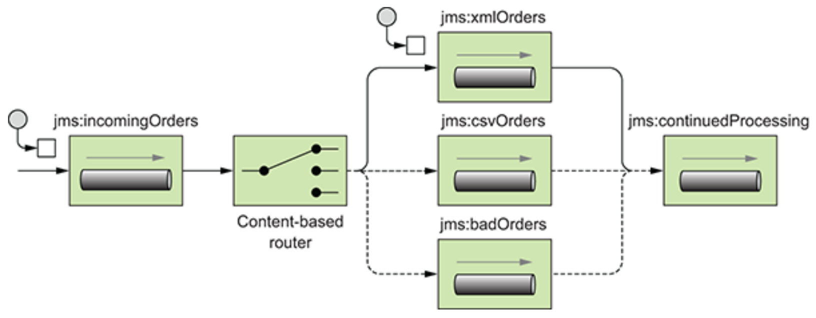


Figure 2.11 By using the `end` method, you can route messages to a destination after the CBR.

You can also control what destinations are final in the `choice` block. For instance, you may not want bad orders continuing through the rest of the route. You'd like them to be routed to the `badOrders` queue and stop there. In that case, you 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 be sent only to the `badOrders` queue—not to the `continuedProcessing` queue.

Using the XML DSL, this route looks a bit different:

```
<route>
  <from uri="jms:incomingOrders"/>
  <choice>
    <when>
      <simple>${header.CamelFileName} ends with '.xml'</simple>
      <to uri="jms:xmlOrders"/>
    </when>
    <when>
      <simple>${header.CamelFileName} regex '^.*(csv|csl)$'</simple>
      <to uri="jms:csvOrders"/>
    </when>
    <otherwise>
```

```
        <to uri="jms:badOrders"/>
        <stop/>
      </otherwise>
    </choice>
    <to uri="jms:continuedProcessing"/>
  </route>
```

Note that you don't have to use an `end()` call to end the choice block because XML requires an explicit *end block* in the form of the closing element `</choice>`.

2.6.2 Using message filters

Rider Auto Parts now has a new issue: its QA department has expressed the need to be able to send test orders into the live web front end of the order system. Your current solution would accept these orders as real and send them to the internal systems for processing. You've 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 you need is a solution that will discard these test messages while still operating on the real orders.

The Message Filter EIP, shown in figure 2.12, provides a nice way of dealing with this kind of problem. Incoming messages pass through the filter only if a certain condition is met. Messages failing the condition are dropped.

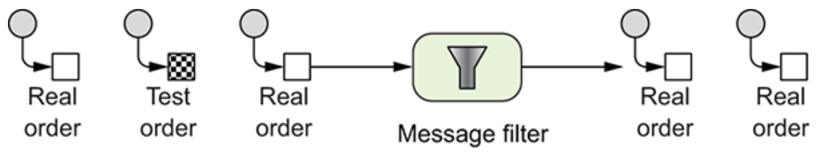


Figure 2.12 A message filter allows you to filter out uninteresting messages based on a certain condition. In this case, test messages are filtered out.

Let's see how to implement this using Camel. Recall that the web front end that Rider Auto Parts uses sends orders only in the XML format, so you can place this filter after the `xmlOrders` queue, where all orders are XML. Test messages have an extra `test` attribute set, so you 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 OrderRouterWithFilterTest.java, which is included with the chapter2/filter project in the book's source distribution. The filter looks like this:

```
from("jms:xmlOrders")
     .filter(xpath("/order[not(@test)]"))
     .log("Received XML order: ${header.CamelFileName}")
     .to("mock:xml");
```

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

```
mvn test -Dtest=OrderRouterWithFilterTest
```

This outputs the following on the command line:

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

You'll receive only one message after the filter because the test message was filtered out.

You may have noticed that this example filters out the test message with an XPath expression. XPath expressions are useful for creating conditions based on XML payloads. In this case, the expression will evaluate to `true` for orders that don't have the `test` attribute.

A message filter route in the XML DSL looks like this:

```
<route>
  <from uri="jms:xmlOrders"/>
  <filter>
    <xpath>/order[not(@test)]</xpath>
    <log message="Received XML order: ${header.CamelFileName}"/>
    <to uri="mock:xml"/>
  </filter>
</route>
```

To run the XML version of the example, execute the following Maven command on the command line:

```
mvn test -Dtest=SpringOrderRouterWithFilterTest
```

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

2.6.3 Using multicasting

Often in enterprise applications you'll need to send a copy of a message to several destinations for processing. When the list of destinations is known ahead of time and is static, you can add an element to the 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're first sent to accounting for validation of customer standing and then to production for manufacture. A bright new manager has suggested improving 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 okay from accounting. You've been asked to implement this change to the system.

Using a multicast, you could envision the solution shown in figure 2.13.

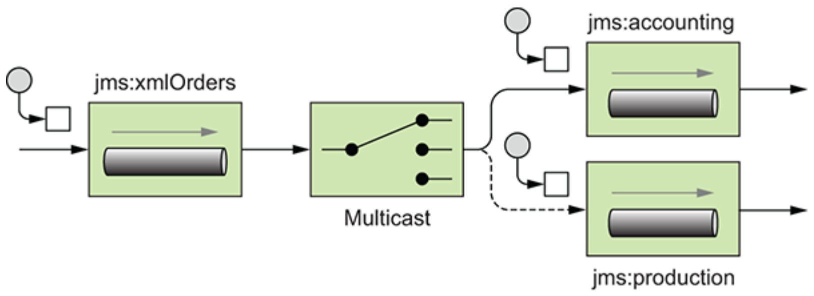


Figure 2.13 A multicast sends a message to numerous specified recipients.

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

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

The equivalent route in XML DSL is as follows:

```
<route>
  <from uri="jms:xmlOrders"/>
  <multicast>
    <to uri="jms:accounting"/>
    <to uri="jms:production"/>
  </multicast>
</route>
```

To run this example, go to the chapter2/multicast directory in the book's source code and run these commands:

```
mvn clean test -Dtest=OrderRouterWithMulticastTest
mvn clean test -Dtest=SpringOrderRouterWithMulticastTest
```

You should see the following output on the command line:

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

These 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.

TIP For dealing with responses from services invoked in a multicast, an aggregator is used. See more about aggregation in chapter 5.

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 you want to send them in parallel?

Using parallel multicasting

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

```
from("jms:xmlOrders")
    .multicast().parallelProcessing()
```

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

This sets up the multicast to distribute messages to the destinations in parallel. Under the hood, a thread pool is used to manage threads. This can be replaced or configured as you see fit. For more information on the Camel threading model and thread pools, see chapter 13.

The equivalent route in XML DSL is as follows:

```
<route>
  <from uri="jms:xmlOrders"/>
  <multicast parallelProcessing="true">
    <to uri="jms:accounting"/>
    <to uri="jms:production"/>
  </multicast>
</route>
```

The main difference from the Java DSL is that the methods used to set flags such as `parallelProcessing` in the Java DSL are now attributes on the `multicast` element. To run this example, go to the chapter2/multicast directory in the book's source code and run these commands:

```
mvn clean test -Dtest=OrderRouterWithParallelMulticastTest
mvn clean test -Dtest=SpringOrderRouterWithParallelMulticastTest
```

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 fails. What do you do in that 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 might take longer to track down the order from production and bill the customer. To solve this problem, you can take advantage of the `stopOnException` feature of the multicast. When enabled, this feature will stop the multicast on the first exception caught, so you can take any necessary action.

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

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

from("direct:accounting")
    .throwException(Exception.class, "I failed!")
    .log("Accounting received order: ${header.CamelFileName}")
    .to("mock:accounting");

from("direct:production")
    .log("Production received order: ${header.CamelFileName}")
    .to("mock:production");
```

To handle the exception coming back from this route, you'll need to use Camel's error-handling facilities, which are described in detail in chapter 11.

TIP Take care when using `stopOnException` with asynchronous messaging. In our example, the exception could have happened after the message had been consumed by both the accounting and production queues, nullifying the `stopOnException` effect. In our test case, we decided to use synchronous direct endpoints, which would allow us to test this feature of the multicast.

When using the XML DSL, this route looks a little different:

```
<route>
  <from uri="jms:xmlOrders"/>
  <multicast stopOnException="true">
    <to uri="direct:accounting"/>
    <to uri="direct:production"/>
  </multicast>
</route>

<route>
  <from uri="direct:accounting"/>
```

```
    <throwException exceptionType="java.lang.Exception" message="I failed!
    <log message="Accounting received order: ${header.CamelFileName}"/>
    <to uri="mock:accounting"/>
  </route>
```

To run this example, go to the chapter2/multicast directory in the book’s source code and run these commands:

```
mvn clean test -Dtest=OrderRouterWithMulticastSOETest
mvn clean test -Dtest=SpringOrderRouterWithMulticastSOETest
```

Now you know how to multicast messages in Camel, but you may be thinking that this seems like a static solution, because changing the destinations means changing the route. Let’s see how to make sending to multiple recipients more dynamic.

2.6.4 Using recipient lists

In the previous section, you implemented a new manager’s suggestion to parallelize the accounting and production queues so orders could be processed more quickly. Rider Auto Parts’ top-tier customers first noticed the problem with this approach: now that all orders are going directly into production, top-tier customers aren’t getting priority over the smaller customers. Their orders are taking longer, and they’re losing business opportunities. Management suggested immediately going back to the old scheme, but you suggested a simple solution to the problem: by parallelizing only top-tier customers’ orders, all other orders would have to go to accounting first, thereby not bogging down production.

This solution can be realized by using the Recipient List EIP. As shown in figure 2.14, a recipient list first inspects the incoming message, then generates a list of desired recipients based on the message content, and sends the message to those recipients. A recipient is specified by an endpoint URI. Note that the recipient list is different from the multicast because the list of recipients is dynamic.

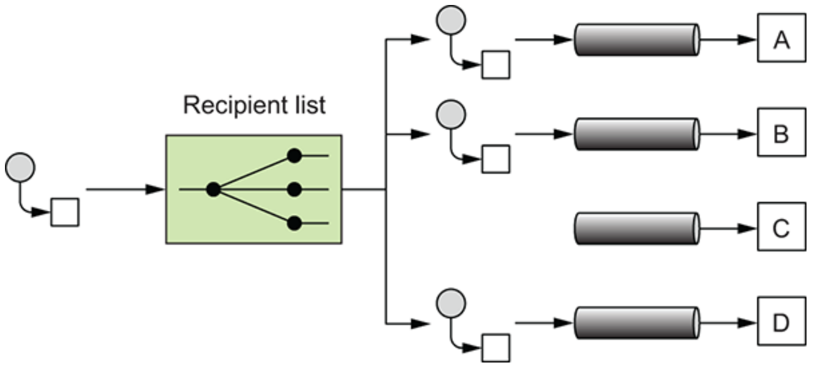


Figure 2.14 A recipient list inspects the incoming message and determines a list of recipients based on the content of the message. In this case, the message contains a list of destinations such as A, B, D. So Camel sends the message to only the A, B, and D destinations. The next message could contain a different set of destinations.

Camel provides a `recipientList` method for implementing the Recipient List EIP. For example, the following route takes the list of recipients from a header named `recipients`, where each recipient is separated from the next by a comma:

```
from("jms:xmlOrders")
    .recipientList(header("recipients"));
```

This is useful if you already have some information in the message that can be used to construct the destination names—you could use an expression to create the list. In order for the recipient list to extract meaningful endpoint URIs, the expression result must be iterable. Values that will work are `java.util.Collection`, `java.util.Iterator`, `java.util.Iterable`, Java arrays, `org.w3c.dom.NodeList`, and, as shown in the example, a `String` with comma-separated values.

In the Rider Auto Parts situation, the message doesn't contain that list. You need some way of determining whether the message is from a top-tier customer. A simple solution could be to call out to a custom Java bean to do this:

```
from("jms:xmlOrders")
    .setHeader("recipients", method(RecipientsBean.class, "recipients"))
    .recipientList(header("recipients"));
```

Here `RecipientsBean` is a simple Java class as follows:

```
public class RecipientsBean {
    public String[] recipients(@XPath("/order/@customer") String custome
        if (isGoldCustomer(customer)) {
            return new String[]{"jms:accounting", "jms:production"};
        } else {
            return new String[]{"jms:accounting"};
        }
    }

    private boolean isGoldCustomer(String customer) {
        return customer.equals("honda");
    }
}
```

The `RecipientsBean` class returns `"jms:accounting,` `jms:production"` only if the customer is at the gold level of support. The check for gold-level support here is greatly simplified; ideally, you'd query a database for this check. Any other orders will be routed only to accounting, which will send them to production after the checks are complete.

The XML DSL version of this route follows a similar layout:

```
  <bean id="recipientsBean" class="camelinaction.RecipientsBean"/>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
   <route>
     <from uri="jms:xmlOrders"/>
     <setHeader headerName="recipients">
       <method ref="recipientsBean" method="recipients"/>
     </setHeader>
     <recipientList>
       <header>recipients</header>
     </recipientList>
   </route>
 </camelContext>
```

The `RecipientsBean` is loaded as a Spring bean and given the name `recipientsBean`, which is then referenced in the `method` element by using the `ref` attribute.

Camel also supports a way of implementing a recipient list without using the exchange and message APIs.

RECIPIENT LIST ANNOTATION

Rather than using the `recipientList` method in the DSL, you can add a `@RecipientList` annotation to a method in a plain Java class (a Java bean). This annotation tells Camel that the annotated method should be used to generate the list of recipients from the exchange. This behavior gets invoked, however, only if the class is used with Camel's bean integration.

For example, replacing the custom bean you used in the previous section with an annotated bean results in a greatly simplified route:

```
from("jms:xmlOrders").bean(AnnotatedRecipientList.class);
```

Now all the logic for calculating the recipients and sending out messages is captured in the `AnnotatedRecipientList` class, which looks like this:

```
public class AnnotatedRecipientList {
    @RecipientList
    public String[] route(@XPath("/order/@customer") String customer) {
        if (isGoldCustomer(customer)) {
            return new String[]{"jms:accounting", "jms:production"};
        } else {
            return new String[]{"jms:accounting"};
        }
    }

    private boolean isGoldCustomer(String customer) {
        return customer.equals("honda");
    }
}
```

Notice that the return type of the bean is a list of the desired recipients. Camel will take this list and send a copy of the message to each destination in the list.

The XML DSL version of this route follows a similar layout:

```
<bean id="annotatedRecipientList"
       class="camelinaction.AnnotatedRecipientList"/>
```

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="jms:xmlOrders"/>
    <bean ref="annotatedRecipientList"/>
  </route>
</camelContext>
```

One nice thing about implementing the recipient list this way is that it's entirely separated from the route, which makes it a bit easier to read. You also have access to Camel's bean-binding annotations, which allow you to extract data from the message by using expressions, so you don't have to manually explore the exchange. This example uses the `@XPath` bean-binding annotation to grab the customer attribute of the order element in the body. We cover these annotations in chapter 4, which is all about using beans. To run this example, go to the chapter2/recipientlist directory in the book's source code and run the command for either the Java or XML DSL case:

```
mvn clean test -Dtest=OrderRouterWithRecipientListAnnotationTest
mvn clean test -Dtest=SpringOrderRouterWithRecipientListAnnotationTest
```

This outputs the following on the command line:

```
Accounting received order: message1.xml
Production received order: message1.xml
Accounting received order: message2.xml
```

Why do you get this output? Well, you had the following two orders in the src/data directory:

- message1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<order name="motor" amount="1000" customer="honda"/>
```

- message2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<order name="motor" amount="2" customer="joe's bikes"/>
```

The first message is from a gold customer, according to the Rider Auto Parts rules, so it was routed to both accounting and production. The second order is from a smaller customer, so it went to accounting for verification of the customer's credit standing.

What this system lacks now is a way to inspect these messages as they're flowing through the route, rather than waiting until they reach the end. Let's see how a wire tap can help.

2.6.5 Using the wireTap method

Often in enterprise applications, inspecting messages as they flow through a system is useful and necessary. For instance, when an order fails, you need a way to look at which messages were received to determine the cause of the failure.

You could use a simple processor, as you've done before, to output information about an incoming message to the console or append it to a file. Here's a processor that outputs the message body to the console:

```
from("jms:incomingOrders")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            System.out.println("Received order: " +
                exchange.getIn().getBody());
        }
    });
```

This is fine for debugging purposes, but it's a poor solution for production use. What if you wanted the message headers, exchange properties, or other data in the message exchange? Ideally, you could copy the whole incoming exchange and send that to another channel for auditing. As shown in figure 2.15, the Wire Tap EIP defines such a solution.

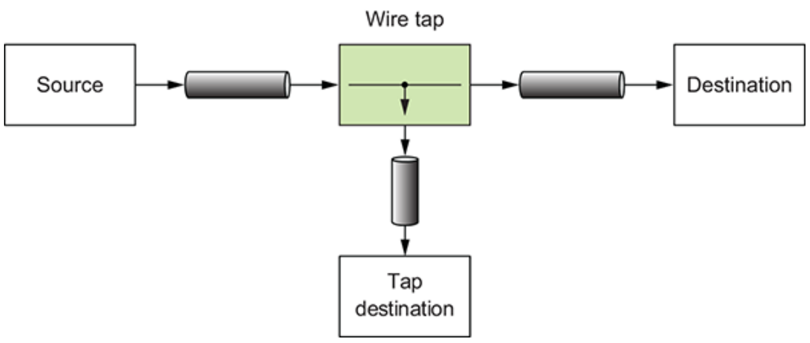


Figure 2.15 A wire tap is a fixed recipient list that sends a copy of a message traveling from a source to a destination to a secondary destination.

By using the `wireTap` method in the Java DSL, you can send a copy of the exchange to a secondary destination without affecting the behavior of the rest of the route:

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

The preceding code sends a copy of the exchange to the `orderAudit` queue, and the original exchange continues on through the route, as if you hadn't used a wire tap at all. Camel doesn't wait for a response from the wire tap because the wire tap sets the message exchange pattern (MEP) to `InOnly`. The message will be sent to the `orderAudit` queue in a fire-and-forget fashion—it won't wait for a reply.

In the XML DSL, you can configure a wire tap just as easily:

```
<route>
  <from uri="jms:incomingOrders"/>
  <wireTap uri="jms:orderAudit"/>
  ...
```

To run this example, go to the chapter2/wiretap directory in the book's source code and run these commands:

```
mvn clean test -Dtest=OrderRouterWithWireTapTest
mvn clean test -Dtest=SpringOrderRouterWithWireTapTest
```

What can you do with a tapped message? Numerous things could be done at this point:

- You could print the information to the console as you did before. This is useful for simple debugging purposes.
- You could save the message in a persistent store (in a file or database) for retrieval later.

The wire tap is a useful monitoring tool, but it leaves most of the work up to you. We'll discuss some of Camel's more powerful tracing and auditing tools in chapter 16.

2.7 Summary and best practices

In this chapter, we've covered probably the most prominent ability of Camel: routing messages. By now you should know how to create routes in either the Java or XML DSL and know the differences in their configuration. You should also have a good grasp of when to apply several EIP implementations in Camel and how to use them. With this knowledge, you can create Camel applications that do useful tasks.

Here are some of the key concepts you should take away from this chapter:

- *Routing occurs in many aspects of everyday life*—Whether you're surfing the internet, doing online banking, or booking a flight or hotel room, messages are being routed behind the scenes via some sort of router.
- *Use Apache Camel for routing messages*—Camel is primarily a message router that allows you to route messages from and to a variety of transports and APIs.
- *Camel's DSLs are used to define routing rules*—The Java DSL allows you to write in the popular Java language, which gives you autocompletion of terms in most IDEs. It also allows you to use the full power of the Java language when writing routes. It's considered the main DSL in Camel. The XML DSL allows you to write routing rules without any Java code at all.

- *The Java DSL and Spring* `CamelContext` *are a powerful combination*—Section 2.4.3 described our favorite way to write Camel applications, which is to boot up `CamelContext` in Spring and write routing rules in Java DSL `RouteBuilder`s. This gives you the best of both: the most expressive DSL that Camel has in the Java DSL, and a more feature-rich and standard container in the Spring `CamelContext`.
- *Use enterprise integration patterns (EIPs) to solve integration and routing problems*—EIPs are like design patterns from object-oriented programming, but for the enterprise integration world.
- *Use Camel's built-in EIP implementations rather than creating your own*—Camel implements most EIPs as easy-to-use DSL terms, which allows you to focus on the business problem rather than the integration architecture.

The coming chapters build on this foundation to show you things like data transformation, using beans, using more advanced EIPs, sending data over other transports, and more. In the next chapter, you'll look at how Camel makes data transformation a breeze.