5

Enterprise integration patterns

This chapter covers

- The Aggregator EIP
- The Splitter EIP
- The Routing Slip EIP
- The Dynamic Router EIP
- The Load Balancer EIP

Today's businesses aren't run on a single monolithic system, and most businesses have a full range of disparate systems. There's an ever-increasing demand for those systems to integrate with each other and with external business partners and government systems.

Let's face it: integration is hard. To help deal with its complexity, enterprise integration patterns (EIPs) have become the standard way to describe, document, and implement complex integration problems. We explain the patterns we discuss in this book, but to learn more about them and others, see the Enterprise Integration Patterns website and the associated book: www.enterpriseintegrationpatterns.com.

5.1 Introducing enterprise integration patterns

Apache Camel implements EIPs, and because the EIPs are essential building blocks in the Camel routes, you'll bump into EIPs throughout this book, starting in chapter 2. It would be impossible for this book to cover all the EIPs Camel supports, which currently total about 70 patterns. This chapter is devoted to covering five of the most powerful and feature-rich patterns, listed in table 5.1.

Table 5.1 EIPs covered in this chapter

Pattern	Summary
Aggregator	Used to combine results of individual but related messages into a single outgoing message. You can view this as the *reverse* of the Splitter pattern. This pattern is covered in section 5.2.
Splitter	Used to split a message into pieces that are routed separately. This pattern is covered in section 5.3.
Routing Slip	Used to route a message in a series of steps; the sequence of steps isn't known at design time and may vary for each message. This pattern is covered in section 5.4.
Dynamic Router	Used to route messages with a dynamic router dictating where the message goes. This pattern is covered in section 5.5.
Load Balancer	Used to balance the load to a given endpoint by using a variety of balancing policies. This pattern is covered in section 5.6.

Let's look at these patterns in more detail.

5.1.1 The Aggregator and Splitter EIPs

The first two patterns listed in table 5.1 are related. The Splitter can split a single message into multiple submessages, and the Aggregator can combine those submessages back into a single message. They're opposite patterns.

The EIPs allow you to build patterns *LEGO style*, which means that patterns can be combined together to form new patterns. For example, you can combine the Splitter and the Aggregator into what is known as the Composed Message Processor EIP, as illustrated in figure 5.1.

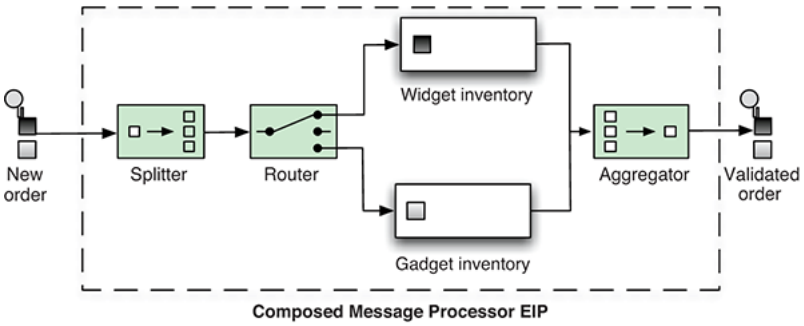

Figure 5.1 The Composed Message Processor EIP splits up the message, routes the submessages to the appropriate destinations, and reaggregates the response back into a single message.

The Aggregator EIP is likely the most sophisticated and most advanced EIP implemented in Camel. It has many use cases, such as aggregating incoming bids for auctions or throttling stock quotes.

5.1.2 The Routing Slip and Dynamic Router EIPs

A question that's often asked on the Camel user mailing list is how to route messages dynamically. The answer is to use EIPs such as Recipient List, Routing Slip, and Dynamic Router. Chapter 2 covered Recipient List, and this chapter will show you how to use the Routing Slip and Dynamic Router patterns.

5.1.3 The Load Balancer EIP

The EIP book doesn't list the Load Balancer, which is a pattern implemented in Camel. Suppose you route PDF messages to network printers, and those printers come and go online. You can use the Load Balancer to send the PDF messages to another printer if one printer is unresponsive.

That covers the five EIPs covered in this chapter. It's now time to look at the first one in detail, the Aggregator EIP.

5.2 The Aggregator EIP

The Aggregator EIP is important and complex, so we'll cover it thoroughly. The Aggregator combines many related incoming messages into a single aggregated message, as illustrated in figure 5.2.

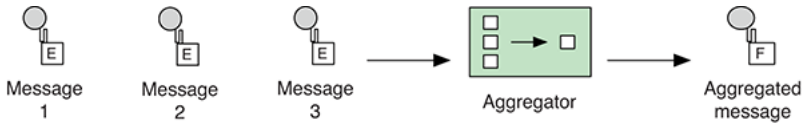

Figure 5.2 The Aggregator stores incoming messages until it receives a complete set of related messages. Then the Aggregator publishes a single message distilled from the individual messages.

The Aggregator receives a stream of messages and identifies messages that are related, which are then aggregated into a single combined message. After a completion condition occurs, the aggregated message is sent to the output channel for further processing. The next section covers how this process works in detail.

EXAMPLE USES OF AGGREGATOR

The Aggregator EIP supports many use cases, such as the loan broker example from the EIP book, in which brokers send loan requests to multiple banks and aggregate the replies to determine the *best deal*.

You could also use the Aggregator in an auction system to aggregate current bids. Also imagine a stock market system that continuously receives a stream of stock quotes, and you want to throttle this to publish the latest quote every five seconds. This can be done by using the Aggregator to choose the latest message and thus trigger a completion every five seconds.

When using the Aggregator, you have to pay attention to the following three settings, which must be configured. Failure to do so will cause Camel to fail on startup and report an error regarding the missing configuration:

- *Correlation identifier*—An `Expression` that determines which incoming messages belong together
- *Completion condition*—A `Predicate` or time-based condition that determines when the result message should be sent
- *Aggregation strategy*—An `AggregationStrategy` that specifies how to combine the messages into a single message

In this section, you'll look at a simple example that will aggregate messages containing alphabetic characters, such as *A*, *B*, and *C*. This will keep things simple, making it easier to follow what's going on. The Aggregator

is equally equipped to work with big loads, but that can wait until we've covered the basic principles.

5.2.1 Using the Aggregator EIP

Suppose you want to collect any three messages and combine them. Given three messages containing *A*, *B*, and *C*, you want the aggregator to output a single message containing *ABC*.

Figure 5.3 shows how this works. When the first message with correlation identifier 1 arrives, the aggregator initializes a new aggregate and stores the message inside the aggregate. In this example, the completion condition is the aggregation of three messages, so the aggregate isn't yet complete. When the second message with correlation identifier 1 arrives, the EIP adds it to the already existing aggregate. The third message specifies a different correlation identifier value of 2, so the aggregator starts a new aggregate for that value. The fourth message relates to the first aggregate (identifier 1), so the aggregate has now aggregated three messages, and the completion condition is fulfilled. As a result, the aggregator marks the aggregate as complete and publishes the resulting message.

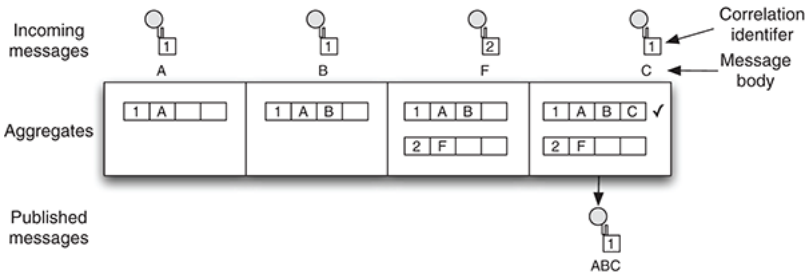

Figure 5.3 The Aggregator EIP in action, with partial aggregated messages updated with arriving messages

As mentioned before, three configurations are in play when using the Aggregator EIP: correlation identifier, completion condition, and aggregation strategy. To understand how these three are specified and how they work, let's start with the example of a Camel route in the Java DSL (with the configurations in bold):

```
public void configure() throws Exception {
    from("direct:start")
        .log("Sending ${body} with correlation key ${header.myId}")
        .aggregate(header("myId"), new MyAggregationStrategy())
            .completionSize(3)
```

```
            .log("Sending out ${body}")
            .to("mock:result");
```

The correlation identifier is `header("myId")`, and it's a Camel `Expression`. It returns the header with the key `myId`. The second configuration element is the `AggregationStrategy`, which is a class. We'll cover this class in more detail in a moment. Finally, the completion condition is based on size (there are seven kinds of completion conditions, listed in table 5.3). It states that when three messages have been aggregated, the completion should trigger.

The same example in XML is as follows:

```
<bean id="myAggregationStrategy"
      class="camelinaction.MyAggregationStrategy"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <log message="Sending ${body} with key ${header.myId}"/>
        <aggregate strategyRef="myAggregationStrategy" completionSize="3
            <correlationExpression>
                <header>myId</header>
            </correlationExpression>
            <log message="Sending out ${body}"/>
            <to uri="mock:result"/>
        </aggregate>
    </route>
</camelContext>
```

The XML snippet is a little different from the Java DSL because you define `AggregationStrategy` by using the `strategyRef` attribute on the `<aggregate>` tag. This refers to a `<bean>`, which is listed in the top of the XML file. The completion condition is also defined as a `completionSize` attribute. The most noticeable difference is the way the correlation identifier is defined. In XML, it's defined using the `<correlationExpression>` tag, which has a child tag that includes the `Expression`.

The book's source code contains this example in the chapter5/aggregator directory. You can run the examples by using the following Maven goals:

```
mvn test -Dtest=AggregateABCTest
mvn test -Dtest=SpringAggregateABCTest
```

The examples use the following unit-test method:

```
public void testABC() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedBodiesReceived("ABC");
    template.sendBodyAndHeader("direct:start", "A", "myId", 1);
    template.sendBodyAndHeader("direct:start", "B", "myId", 1);
    template.sendBodyAndHeader("direct:start", "F", "myId", 2);
    template.sendBodyAndHeader("direct:start", "C", "myId", 1);
    assertMockEndpointsSatisfied();
}
```

This unit test sends the same messages as shown in figure 5.3—four messages in total. When you run the test, you'll see the output on the console:

```
INFO route1 - Sending A with correlation key 1
INFO route1 - Sending B with correlation key 1
INFO route1 - Sending F with correlation key 2
INFO route1 - Sending C with correlation key 1
INFO route1 - Sending out ABC
```

Notice that the console output matches the sequence in which the messages were aggregated in the example from figure 5.3. As you can see from the console output, the messages with correlation key 1 were completed, because they met the completion condition, which was size based on three messages. The last line of the output shows the published message, which contains the letters *ABC*.

So what happens with the *F* message? Well, its completion condition hasn't been met, so it waits in the aggregator. You could modify the test method to send an additional two messages to complete that second group as well:

```
template.sendBodyAndHeader("direct:start", "G", "myId", 2);
template.sendBodyAndHeader("direct:start", "H", "myId", 2);
```

Let's now turn our focus to how the Aggregator EIP combines the messages, which causes the *A*, *B*, and *C* messages to be published as a single message. This is where the `AggregationStrategy` comes into the picture, because it orchestrates this.

Using AggregationStrategy

The `AggregationStrategy` class is located in the `org.apache.camel.processor.aggregate` package, and it defines a single method:

```
public interface AggregationStrategy {
    Exchange aggregate(Exchange oldExchange, Exchange newExchange);
}
```

If you're having a déjà vu moment, it's most likely because `AggregationStrategy` is also used by the Content Enricher EIP, which we covered in chapter 3.

The following listing shows the strategy used in the previous example.

Listing 5.1 `AggregationStrategy` for merging messages

```
public class MyAggregationStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange
        if (oldExchange == null) {
      return newExchange;   ❶
```

❶ Occurs for a new group

```
        }
    String oldBody = oldExchange.getIn()   ❷
```

❷ Combines message bodies

```
                .getBody(String.class);    ❷
        String newBody = newExchange.getIn()    ❷
                .getBody(String.class);    ❷
        String body = oldBody + newBody;    ❷
        oldExchange.getIn().setBody(body);    ❶
```

---

❸

Replaces message body with combined message bodies

```
            return oldExchange;
        }
    }
```

At runtime, the `aggregate` method is invoked every time a new message arrives. In this example, it'll be invoked four times: one for each arriving message *A*, *B*, *F*, and *C*. To show how this works, table 5.2 lists the invocations as they'd happen.

Table 5.2 Sequence of invocations of `aggregate` method occurring at runtime

Arrived	oldExchange	newExchange	Description
`A`	`null`	`A`	The first message arrives for the first group.
`B`	`A`	`B`	The second message arrives for the first group.
`F`	`null`	`F`	The first message arrives for the second group.
`C`	`AB`	`C`	The third message arrives for the first group.

Notice in table 5.2 that the `oldExchange` parameter is `null` on two occasions. This occurs when a new correlation group is formed (no preexisting messages have arrived with the same correlation identifier). In this situation, you want to return the message as is, because there are no other messages to combine it with ❶.

On the subsequent aggregations, neither parameter is `null`, so you need to merge the data into one `Exchange`. In this example, you grab the message bodies and add them together ❷. Then you replace the existing body in `oldExchange` with the updated body ❸.

NOTE The Aggregator EIP uses synchronization, which ensures that `AggregationStrategy` is thread safe—only one thread is invoking the `aggregate` method at any time. The Aggregator also ensures ordering, which means the messages are aggregated in the same order as they're sent into the Aggregator.

You should now understand the principles of how the Aggregator works. For a message to be published from the Aggregator, a completion condition must have been met. In the next section, we discuss this and review the conditions Camel provides out of the box.

5.2.2 Completion conditions for the Aggregator

Completion conditions play a bigger role in the Aggregator than you might think. Imagine a situation in which a condition never occurs, causing aggregated messages never to be published. For example, suppose the *C* message never arrived in the example in section 5.2.1. To remedy this, you could add a time-out condition that reacts if all messages aren't received within a certain time period.

To cater for that situation and others, Camel provides seven completion conditions, listed in table 5.3. You can mix and match them according to your needs.

Table 5.3 Completion conditions provided by the Aggregator EIP

Condition	Description
`completionSize`	Defines a completion condition based on the number of messages aggregated together. You can either use a fixed value (`int`) or use an `Expression` to dynamically decide a size at runtime.
`completionTimeout`	Defines a completion condition based on an inactivity time-out. This condition triggers if a correlation group has been inactive longer than the specified period. Time-outs are scheduled for each correlation group, so the time-out is individual to each group. You can either use a fixed value (`long`) or an `Expression` to dynamically decide a time-out at runtime. The period is defined in milliseconds. You can't use this condition together with the `completionInterval` .
`completionInterval`	Defines a completion condition based on a scheduled interval. This condition triggers periodically. There's a single scheduled time-out for *all* correlation groups, which causes all groups to complete at the same time. The period (`long`) is defined in milliseconds. You can't use this condition together with the `completionTimeout` .
`completionPredicate`	Defines a completion condition based on whether the `Predicate` matched. See also the `eagerCheckCompletion` option in table 5.5. This condition is enabled automatically if `aggregationStrategy` implements either

Condition	Description
	`Predicate` or `PreCompletionAwareAggregationStrategy`. In the case of `PreCompletionAwareAggregationStrategy`, this gives you the ability to complete the aggregation group on receipt of a new `Exchange` and start a new group with the new `Exchange`.
`completionFromBatchConsumer`	Defines a completion condition that's applicable only when the arriving `Exchange`s are coming from a `BatchConsumer` (http://camel.apache.org/batch-consumer.html). Numerous components support this condition, such as Atom, File, FTP, HBase, Mail, MyBatis, JClouds, SNMP, SQL, SQS, S3, and JPA.
`forceCompletionOnStop`	Defines a completion condition that will complete all correlation groups on shutdown of `CamelContext`.
`aggregateController`	By using `AggregateController`, you can control group completion externally via Java calls into the controller or via JMX.

The Aggregator supports using multiple completion conditions, such as using both the `completionSize` and `completionTimeout` conditions. When using multiple conditions, though, the winner takes all: the completion condition that completes first will result in the message being published.

NOTE The book's source code contains examples in the chapter5/aggregator directory for all conditions; you can refer to them for further details. Also the Aggregator documentation on the Camel website has more details: http://camel.apache.org/aggregator2.

We'll now look at how to use multiple completion conditions.

Using multiple completion conditions

The book's source code contains an example in the chapter5/aggregator directory showing how to use multiple completion conditions. You can run the example by using the following Maven goals:

```
mvn test -Dtest=AggregateXMLTest
mvn test -Dtest=SpringAggregateXMLTest
```

The route in the Java DSL is as follows:

```
import static org.apache.camel.builder.xml.XPathBuilder.xpath;

public void configure() throws Exception {
    from("direct:start")
        .log("Sending ${body}")
        .aggregate(xpath("/order/@customer"), new MyAggregationStrategy(
            .completionSize(2).completionTimeout(5000)
            .log("Sending out ${body}")
            .to("mock:result");
}
```

As you can see from the bold code in the route, using a second condition is just a matter of adding a completion condition.

The same example in XML is shown here:

```
<bean id="myAggregationStrategy"
       class="camelinaction.MyAggregationStrategy"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
```

```
        <log message="Sending ${body}"/>
        <aggregate strategyRef="myAggregationStrategy"
                      completionSize="2" completionTimeout="5000">
            <correlationExpression>
                <xpath>/order/@customer</xpath>
            </correlationExpression>
            <log message="Sending out ${body}"/>
            <to uri="mock:result"/>
        </aggregate>
    </route>
</camelContext>
```

If you run this example, it'll use the following test method:

```
public void testXML() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(2);
    template.sendBody("direct:start",
        "<order name=\"motor\" amount=\"1000\" customer=\"honda\"/>");
    template.sendBody("direct:start",
        "<order name=\"motor\" amount=\"500\" customer=\"toyota\"/>");
    template.sendBody("direct:start",
        "<order name=\"gearbox\" amount=\"200\" customer=\"toyota\"/>");
    assertMockEndpointsSatisfied();
}
```

This example should cause the aggregator to publish two outgoing messages, as shown in the following console output—one for Honda and one for Toyota:

```
09:37:35 - Sending <order name="motor" amount="1000" customer="honda"/>
09:37:35 - Sending <order name="motor" amount="500" customer="toyota"/>
09:37:35 - Sending <order name="gearbox" amount="200" customer="toyota"/
09:37:35 - Sending out
           <order name="motor" amount="500" customer="toyota"/>
           <order name="gearbox" amount="200" customer="toyota"/>
09:37:41 - Sending out
           <order name="motor" amount="1000" customer="honda"/>
```

If you look closely at the test method and the output from the console, you should notice that the Honda order arrived first, but it was the last to

be published. This is because its completion was triggered by the time-out, which was set to 5 seconds. In the meantime, the Toyota order had its completion triggered by the size of two messages, so it was published first.

TIP The Aggregator EIP allows you to use as many completion conditions as you like. But the `completionTimeout` and `completionInterval` conditions can't be used at the same time.

Using multiple completion conditions makes good sense if you want to ensure that aggregated messages eventually get published. For example, the time-out condition ensures that after a period of inactivity, the message will be published. In that regard, you can use the time-out condition as a fallback condition, with the price being that the published message will be only partly aggregated. Suppose you expect two messages to be aggregated into one, but you receive only one message; the next section reveals how you can tell which condition triggered the completion.

Aggregated exchange properties

Camel enriches the published `Exchange` with the completion details listed in table 5.4.

Table 5.4 Properties on the `Exchange` related to aggregation

Property	Type	Descripti
`Exchange.AGGREGATED_SIZE`	`int`	The total number c arrived messages aggregate
`Exchange.AGGREGATED_COMPLETED_BY`	`String`	The condition that triggered the completic Possible values ar `size`, `timeout` `interval` `predicat` `force`, `strategy` and `consumer` The `consumer` value represent the completic from bat consume
`Exchange.AGGREGATED_CORRELATION_KEY`	`String`	The correlatic

Property	Type	Descripti
		identifier a `String`
`Exchange.AGGREGATED_TIMEOUT`	`long`	The time-out in milliseco as set by t completio time-out.
`Exchange.AGGREGATION_COMPLETE_CURRENT_GROUP`	`boolean`	Set this to `true` to complete the curre group.
`Exchange.AGGREGATION_COMPLETE_ALL_GROUPS`	`boolean`	Set this to `true` to complete groups ar ignore th current `Exchange`
`Exchange.AGGREGATION_COMPLETE_ALL_GROUPS_INCLUSIVE`	`boolean`	Set this to `true` to complete groups ar include th current `Exchange`

The information listed in table 5.4 allows you to know how a published aggregated `Exchange` was completed, and how many messages were

combined. For example, you could log which condition triggered the completion by adding this to the Camel route:

```
.log("Completed by ${property.CamelAggregatedCompletedBy}")
```

This information might come in handy in your business logic, when you need to know whether all messages were aggregated. You can tell this by checking the `AGGREGATED_COMPLETED_BY` property, which could contain several values, including `size` and `timeout`. If the value is `size`, all the messages were aggregated; if the value is `timeout`, a time-out occurred, and not all expected messages were aggregated. The Aggregator has additional configuration options that you may need to use. For example, you can specify how it should react when an arrived message contains an invalid correlation identifier.

Additional configuration options

The Aggregator is the most sophisticated EIP implemented in Camel, and table 5.5 lists the additional configuration options you can use to tweak it to fit your needs.

Table 5.5 Additional configuration options available for the Aggregator EIP

Configuration option	Default	Des
`eagerCheckCompletion`	`false`	This option specifies whether to checking means Camel will chec aggregating. By default, Camel c aggregation. This option is used to control ho condition behaves. If the option will use the aggregated `Exchang` incoming `Exchange` will be use
`closeCorrelationKeyOnCompletion`		This option determines whether marked as closed when it's com closed, any subsequent arriving `ClosedCorrelationKeyExcepti` `Integer` parameter that repres recently used (LRU) cache. This Note that this cache is in-memor restarted.
`ignoreInvalidCorrelationKeys`	`false`	This option specifies whether to default, Camel throws a `CamelE` You can suppress this by setting Camel skips the invalid message
`timeoutCheckerExecutorService` / `timeoutCheckerExecutorServiceRef`		Specifies a `ScheduledExecutorS` when checking for timeout-base
`optimisticLocking`	`false`	Turns on optimistic locking of th aggregation repository must imp `org.apache.camel.spi.Optimi`
`optimisticLockRetryPolicy`		Specifies `OptimisticLockRetry` retries occur.

If you want to learn more about the configuration options listed in table 5.5, there are examples for most in the book's source code in the chapter5/aggregator directory. You can run test examples by using the following Maven goals:

```
mvn test -Dtest=AggregateABCEagerTest
mvn test -Dtest=SpringAggregateABCEagerTest
mvn test -Dtest=AggregateABCCloseTest
mvn test -Dtest=SpringAggregateABCCloseTest
mvn test -Dtest=AggregateABCInvalidTest
mvn test -Dtest=SpringAggregateABCInvalidTest
mvn test -Dtest=AggregateABCGroupTest
mvn test -Dtest=SpringAggregateABCGroupTest
mvn test -Dtest=AggregateTimeoutThreadpoolTest
mvn test -Dtest=SpringAggregateTimeoutThreadpoolTest
```

Next, we'll look at implementing aggregation strategies without using any Camel API at all.

Using POJOs for the AggregationStrategy

So far you've seen that you can customize how `Exchanges` are aggregated by implementing `AggregationStrategy` in a custom class. There's a slightly cleaner way of doing this, however, without using Camel APIs at all. As with Camel's bean integration, you can provide the aggregator with a POJO to act as `AggregationStrategy`. Camel handles injecting one or all of the message body, headers, and `Exchange` properties. Taking a look at `AggregationStrategy` used in listing 5.1, you can provide an equivalent POJO version as follows:

```
public class MyAggregationStrategyPojo {
    public String concat(String oldBody, String newBody) {
        if (newBody != null) {
            return oldBody + newBody;
        } else {
            return oldBody;
        }
    }
}
```

As you can see, it's much cleaner than `AggregationStrategy` in listing 5.1. Similarly, though, it has two parameters: one for the existing aggregated message (`oldBody`) and one for the incoming message (`newBody`). If you need either the message headers and/or the `Exchange` properties, you can use method signatures such as these:

```
public String concat(String oldBody, Map oldHeaders,
                     String newBody, Map newHeaders);
public String concat(String oldBody, Map oldHeaders, Map oldProperties,
                     String newBody, Map newHeaders, Map newPropert
```

Notice that you have to add more parameters in pairs and that order is important. The first parameter is the body, the second is the header, and third is exchange properties.

The Camel route looks different from before as well. When referencing the preceding `AggregationStrategy` POJO, your route looks like this:

```
import org.apache.camel.util.toolbox.AggregationStrategies;

public void configure() throws Exception {
    from("direct:start")
        .log("Sending ${body} with correlation key ${header.myId}")
        .aggregate(header("myId"), AggregationStrategies.bean(new
       .completionSize(3)
              .log("Sending out ${body}")
            .to("mock:result");
 }
```

As you can see in bold in the preceding code, you use the `AggregationStrategies` utility class to convert the POJO into an `AggregationStrategy`. Many more methods are available in this class as well. For instance, you can pass in a bean reference and select the method you want to use, or even just the class.

The same route in XML is shown here:

```
<bean id="myAggregationStrategy"
      class="camelinaction.MyAggregationStrategyPojo"/>
```

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <log message="Sending ${body} with correlation key ${header.myId
        <aggregate strategyRef="myAggregationStrategy" completionSize="3
            <correlationExpression>
                <header>myId</header>
            </correlationExpression>
            <log message="Sending out ${body}"/>
            <to uri="mock:result"/>
        </aggregate>
    </route>
</camelContext>
```

If you want to try this for yourself, examples are provided in the book's source code, in the chapter5/aggregator directory. You can run test examples by using the following Maven goals:

```
mvn test -Dtest=AggregatePojoTest
mvn test -Dtest=SpringAggregatePojoTest
```

In the next section, we'll look at solving the problems with persistence. The Aggregator, by default, uses an in-memory repository to hold the current in-progress aggregated messages, and those messages will be lost if the application is stopped or the server crashes. To remedy this, you need to use a persisted repository.

5.2.3 Using persistence with the Aggregator

The Aggregator is a stateful EIP because it needs to store the in-progress aggregates until completion conditions occur and the aggregated message can be published. By default, the Aggregator will keep state in memory only. If the application is shut down or the host container crashes, the state will be lost.

To remedy this problem, you need to store the state in a persistent repository. Camel provides a pluggable feature so you can use a repository of your choice. This comes in three flavors:

- `AggregationRepository` —An interface that defines the general operations for working with a repository, such as adding data to and remov-

ing data from it. By default, Camel uses `MemoryAggregationRepository`, which is a memory-only repository.

- `RecoverableAggregationRepository`—An interface that defines additional operations supporting recovery. Camel provides several such repositories out of the box, including `JdbcAggregationRepository`, `CassandraAggregationRepository`, `LevelDBAggregationRepository`, and `HazelcastAggregationRepository`. We cover recovery in section 5.2.4.
- `OptimisticLockingAggregationRepository`—An interface that defines additional operations supporting optimistic locking. The `MemoryAggregationRepository` and `JdbcAggregationRepository` repositories implement this interface.

ABOUT LEVELDB

LevelDB is a lightweight and embeddable key-value storage library. It allows Camel to provide persistence for various Camel features, such as the Aggregator.

You can find more information about LevelDB at its website: http://github.com/google/leveldb.

Now we'll look at how to use LevelDB as a persistent repository.

Using camel-leveldb

To demonstrate how to use LevelDB with the Aggregator, we'll return to the *ABC* example. In essence, all you need to do is instruct the Aggregator to use `LevelDB-AggregationRepository` as its repository.

First, though, you must set up LevelDB, which is done as follows:

```
AggregationRepository myRepo = new
    LevelDBAggregationRepository("myrepo", "data/myrepo.dat");
```

Or, in XML, you do this:

```
<bean id="myRepo"
       class="org.apache.camel.component.leveldb.LevelDBAggregationRepos
    <property name="repositoryName" value="myrepo"/>
```

```
    <property name="persistentFileName" value="data/myrepo.dat"/>
</bean>
```

As you can see, this creates a new instance of `LevelDBAggregationRepository` and provides two parameters: the repository name, which is a symbolic name, and the physical filename to use as persistent storage. The repository name must be specified because you can have multiple repositories in the same file.

TIP You can find information about the additional supported options for the LevelDB component at the Camel website: http://camel.apache.org/leveldb.

To use `LevelDBAggregationRepository` in the Camel route, you can instruct the Aggregator to use it, as shown in the following listing.

Listing 5.2 Using LevelDB with Aggregator in Java DSL

```
AggregationRepository myRepo =
    new LevelDBAggregationRepository("myrepo", "data/myrepo.dat");

from("file://target/inbox")
    .log("Consuming ${file:name}")
    .convertBodyTo(String.class)
    .aggregate(constant(true), new MyAggregationStrategy())
        .aggregationRepository(myRepo)
        .completionSize(3)
        .log("Sending out ${body}")
        .to("mock:result");
```

The next listing shows the same example in XML.

Listing 5.3 Using LevelDB with Aggregator in XML

```
<bean id="myAggregationStrategy"
      class="camelinaction.MyAggregationStrategy"/>
<bean id="myRepo    (1)
```

❶

LevelDB persistent repository

```
        class="org.apache.camel.component.leveldb.LevelDBAggregationRepos
    <property name="repositoryName" value="myrepo"/>
    <property name="persistentFileName" value="data/myrepo.dat"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file://target/inbox"/>
        <log message="Consuming ${file:name}"/>
        <convertBodyTo type="java.lang.String"/>
        <aggregate strategyRef="myAggregationStrategy" completionSize="3
                    aggregationRepositoryRef="myRepo">
            <correlationExpression>
                <constant>true</constant>
            </correlationExpression>
            <log message="Sending out ${body}"/>
        </aggregate>
    </route>
</camelContext>
```

As you can see, a Spring `bean` tag is defined with the ID `myRepo` ❶, which sets up the persistent `AggregationRepository`. The name for the repository and the filename are configured as properties on the `bean` tag. In the Camel route, you then refer to this repository by using the `aggregationRepositoryRef` attribute on the `aggregate` tag.

Running the example

The book's source code contains this example in the chapter5/aggregator directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=AggregateABCLevelDBTest
mvn test -Dtest=SpringAggregateABCLevelDBTest
```

To demonstrate how the persistence store works, the example will start up and run for 20 seconds. In that time, you can copy files in the

target/inbox directory and have those files consumed and aggregated. On every third file, the Aggregator will complete and publish a message.

The example displays instructions on the console about how to do this:

```
Copy 3 files to target/inbox to trigger the completion
Files to copy:
  copy src/test/resources/a.txt target/inbox
  copy src/test/resources/b.txt target/inbox
  copy src/test/resources/c.txt target/inbox
Sleeping for 20 seconds
You can let the test terminate (or press ctrl + c) and then start it aga
Which should let you be able to resume.
```

For instance, if you copy the first two files and then let the example terminate, you'll see the following:

```
cd chapter5/aggregator
chapter5/aggregator$ cp src/test/resources/a.txt target/inbox
chapter5/aggregator$ cp src/test/resources/b.txt target/inbox
```

The console should indicate that it consumed two files and was shut down:

```
2017-05-07 12:29:33,714 [ #1 - file://target/inbox]
    INFO route1 - Consuming file a.txt
2017-05-07 12:29:35,235 [ #1 - file://target/inbox]
    INFO route1 - Consuming file b.txt
...
2017-05-07 12:29:43,224 [ main]
    INFO DefaultCamelContext - Apache Camel 2.20.1 (CamelContext:
    camel-1) is shutdown in 0.022 seconds
```

The next time you start the example, you can resume where you left off, and copy the last file:

```
chapter5/aggregator$ cp src/test/resources/c.txt target/inbox
```

Then the Aggregator should complete and publish the message:

```
2017-05-07 12:40:35,069 [ main]
    INFO  LevelDBAggregationRepository   - On startup there are 1
    aggregate exchanges (not completed) in repository: myrepo
...
2017-05-07 12:40:38,589 [ #1 - file://target/inbox]
    INFO route1 - Consuming file c.txt
2017-05-07 12:40:38,606 [ #1 - file://target/inbox]
    INFO route1 - Sending out ABC
```

Notice that it logs on startup the number of exchanges that are in the persistent repository. This example has one existing `Exchange` on startup.

Now you've seen the persistent Aggregator in action. Let's move on to look at using recovery with the Aggregator, which ensures that published messages can be safely recovered and be routed in a transactional way.

5.2.4 Using recovery with the Aggregator

The examples covered in the previous section focused on ensuring that messages are persisted during aggregation. But there's another way messages may be lost: messages that have been published (sent out) from the Aggregator could fail during routing as well.

To remedy this problem, you could use one of these two approaches:

- *Camel error handlers*—These provide redelivery and dead letter channel capabilities. We cover them in chapter 11.
- `RecoverableAggregationRepository`—This interface *extends* `AggregationRepository` and offers the recovery, redelivery, and dead letter channel features.

Camel error handlers aren't tightly coupled with the Aggregator, so message handling is in the hands of the error handler. If a message repeatedly fails, the error handler can deal with this only by retrying or eventually giving up and moving the message to a dead letter channel.

`RecoverableAggregationRepository`, on the other hand, is tightly integrated into the Aggregator, which allows additional benefits such as using the persistence store for recovery and offering transactional capabilities. It ensures that published messages that fail will be recovered and redelivered. You can think of this as what a JMS broker, such as Apache

ActiveMQ, can do by bumping failed messages back up on the JMS queue for redelivery.

Understanding recovery

To better understand how recovery works, consider the next two figures. Figure 5.4 shows what happens when an aggregated message is being published for the first time and the message fails during processing. This could also be the situation when a server crashes while processing the message.

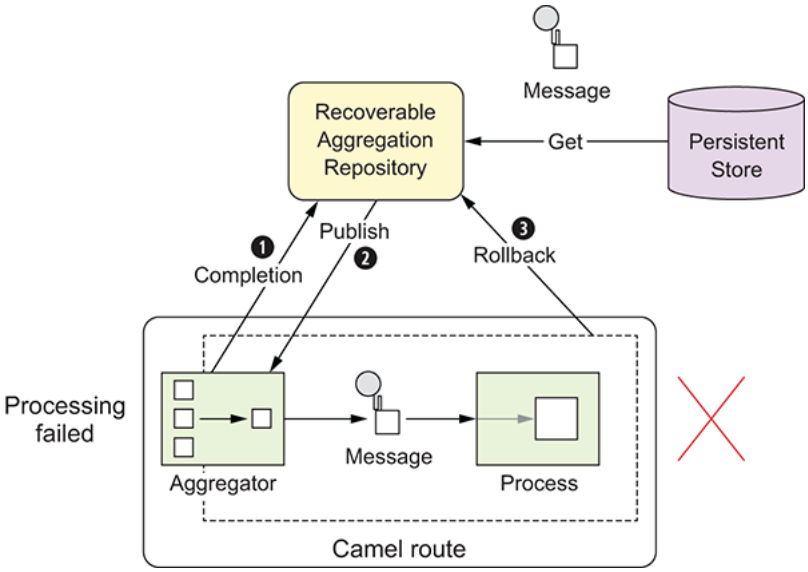

Figure 5.4 An aggregated message is completed ❶, it's published from the Aggregator ❷, and processing fails ❸, so the message is rolled back.

An aggregated message is complete, so the Aggregator signals ❶ this to the `RecoverableAggregationRepository`, which fetches the aggregated message to be published ❷. The message is then routed in Camel—but suppose it fails during routing ❸? A signal is sent from the Aggregator to `RecoverableAggregationRepository`, which can act accordingly.

Now imagine the same message is recovered and redelivered, as shown in figure 5.5.

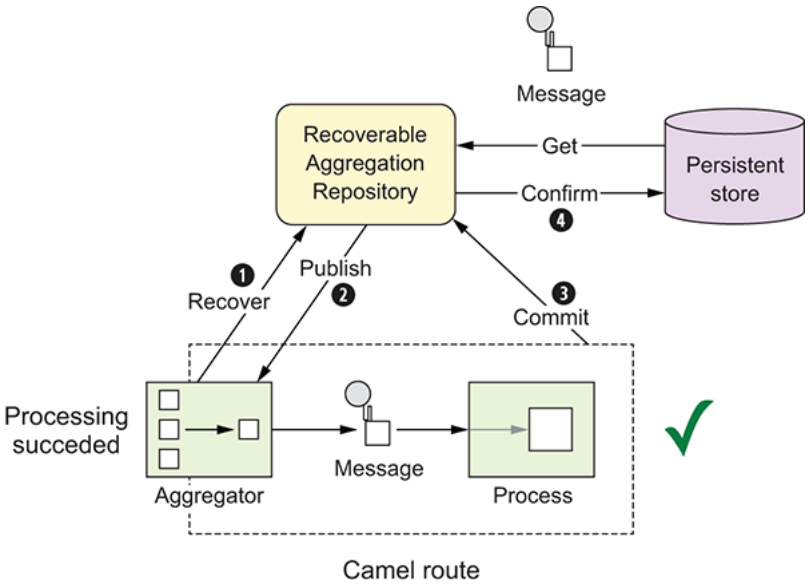

Figure 5.5 The Aggregator recovers failed messages ❶, which are published again ❷, and this time the messages completed ❸ successfully ❹.

The Aggregator uses a background task, which runs at regular intervals (use `setRecoveryInterval` on the `RecoverableAggregationRepository` to specify this), to scan for previously published messages to be recovered ❶. Any such messages will be republished ❷, and the message will be routed again. This time, the message could be processed successfully, which lets the Aggregator issue a commit ❸. The repository confirms the message ❹, ensuring that it won't be recovered on subsequent scans.

NOTE The transactional behavior provided by `RecoverableAggregationRepository` isn't based on Spring's `TransactionManager` (which we cover in chapter 12). The transactional behavior is based on LevelDB's own transaction mechanism (because we're using the `LevelDBAggregationRepository`).

Running the example

The book's source code contains this example in the chapter5/aggregator directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=AggregateABCRecoverTest
mvn test -Dtest=SpringAggregateABCRecoverTest
```

The example is constructed to fail when processing the published messages, no matter what. Eventually, you'll have to move the message to a dead letter channel.

To use recovery with routes in the Java DSL, you have to set up `LevelDBAggregationRepository` as shown here:

```
LevelDBAggregationRepository levelDB = new
    LevelDBAggregationRepository("myrepo", "data/myrepo.dat");
levelDB.setUseRecovery(true);
levelDB.setMaximumRedeliveries(4);
levelDB.setDeadLetterUri("mock:dead");
levelDB.setRecoveryInterval(3000);
```

In XML, you can set this up as a `<bean>` tag, as follows:

```
<bean id="myRepo"
       class="org.apache.camel.component.leveldb.LevelDBAggregationRepos
    <property name="repositoryName" value="myrepo"/>
    <property name="persistentFileName" value="data/myrepo.dat"/>
    <property name="useRecovery" value="true"/>
    <property name="recoveryInterval" value="3000"/>
    <property name="maximumRedeliveries" value="4"/>
    <property name="deadLetterUri" value="mock:dead"/>
</bean>
```

The options may make sense as you read them now, but you'll revisit them in table 5.7. In this example, the Aggregator will check for messages to be recovered every 3 seconds. To avoid a message being repeatedly recovered, the maximum redeliveries are set to 4. After four failed recovery attempts, the message is exhausted and moved to the dead letter channel. If you omit the maximum redeliveries option, Camel will keep recovering failed messages forever until they can be processed successfully.

If you run the example, you'll notice that the console outputs the failures as stack traces, and at the end you'll see a `WARN` entry that indicates the message has been moved to the dead letter channel:

```
2017-05-07 22:28:18,997 [- AggregateRecoverChecker]
    WARN AggregateProcessor - The recovered exchange is exhausted after
    attempts, will now be moved to dead letter channel: mock:dead
```

We encourage you to try this example and read the comments in the source code to better understand how this works.

The preceding log output identifies the number of redelivery attempts, but how does Camel know this? Camel stores this information on the `Exchange`. Table 5.6 reveals where this information is stored.

Table 5.6 Headers on `Exchange` related to redelivery

Header	Type	Description
`Exchange.REDELIVERY_COUNTER`	`int`	The current redelivery attempt. The counter starts with the value of 1.
`Exchange.REDELIVERY_MAX_COUNTER`	`int`	The maximum redelivery attempts that will be made.
`Exchange.REDELIVERED`	`boolean`	Whether this `Exchange` is being redelivered.
`Exchange.REDELIVERY_EXHAUSTED`	`boolean`	Whether this `Exchange` has attempted all redeliveries and still failed (also known as being exhausted).
`Exchange.REDELIVERY_DELAY`	`long`	Delay in milliseconds before scheduling redelivery.

The information in table 5.6 is available only when Camel performs a recovery. These headers are absent on the regular first attempt. It's only

when a recovery is triggered that these headers are set on the `Exchange`.

Table 5.7 lists the options for `RecoverableAggregationRepository` that are related to recovery.

Table 5.7 `RecoverableAggregationRepository` configuration options related to recovery

Option	Default	Description
`useRecovery`	`true`	Whether recovery is enabled.
`recoveryInterval`	`5000`	How often the recovery background tasks are executed. The value is in milliseconds.
`deadLetterUri`		An optional dead letter channel, where published messages that are exhausted should be sent. This is similar to the `DeadLetterChannel` error handler, which we cover in chapter 11. This option is disabled by default. When in use, the `maximumRedeliveries` option must be configured as well.
`maximumRedeliveries`		A limit that defines when published messages that repeatedly fail are considered exhausted and should be moved to the dead letter URI. This option is disabled by default.

We won't go into more detail regarding the options in table 5.7, as we've already covered an example using them.

This concludes our extensive coverage of the sophisticated and probably most complex EIP implemented in Camel—the Aggregator. In the next section, we'll look at the Splitter pattern.

5.3 The Splitter EIP

Messages passing through an integration solution may consist of multiple elements, such as an order, which typically consists of more than a single line item. Each line in the order may need to be handled differently, so you need an approach that processes the complete order, treating each line item individually. The solution to this problem is the Splitter EIP, illustrated in figure 5.6.

Figure 5.6 The Splitter breaks the incoming message into a series of individual messages.

In this section, we'll teach you all you need to know about the Splitter. You'll start with a simple example and move on from there.

5.3.1 Using the Splitter

Using the Splitter in Camel is straightforward, so let's try a basic example that will split one message into three messages, each containing one of the letters *A*, *B*, and *C*. The following listing shows the example using a Java DSL–based Camel route and a unit test.

Listing 5.4 A basic example of the Splitter EIP

```
public class SplitterABCTest extends CamelTestSupport {
    public void testSplitABC() throws Exception {
        MockEndpoint mock = getMockEndpoint("mock:split");
        mock.expectedBodiesReceived("A", "B", "C");
        List<String> body = new ArrayList<String>();
        body.add("A");
        body.add("B");
        body.add("C");
        template.sendBody("direct:start", body);
        assertMockEndpointsSatisfied();
    }
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
```

```
            public void configure() throws Exception {
                from("direct:start")
                .split(body())  ❷
```

❶ Splits incoming message body

```
                        .log("Split line ${body}")
                        .to("mock:split");
            }
        };
    }
}
```

The test method sets up a mock endpoint that expects three messages to arrive, in the order *A*, *B*, and *C*. Then you construct a single combined message body that consists of a `List` of `String`s containing the three letters. The Camel route will use the Splitter EIP to split up the message body ❶.

If you run this test, the console should log the three messages, as follows:

```
INFO  route1 - Split line A
INFO  route1 - Split line B
INFO  route1 - Split line C
```

When using the Splitter EIP in XML, you have to do this differently because the Splitter uses an `Expression` to return what is to be split.

In the Java DSL, you defined the `Expression` shown in bold:

```
.split(body())
```

Here, `body` is a method available on `RouteBuilder`, which returns an `org.apache. camel.Expression` instance. In XML, you need to do this as shown in bold:

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

```
        <from uri="direct:start"/>
        <split>
            <simple>${body}</simple>
            <log message="Split line ${body}"/>
            <to uri="mock:split"/>
        </split>
    </route>
</camelContext>
```

In XML, you use Camel's expression language, known as Simple (discussed in appendix A), to tell the Splitter that it should split the message body.

The book's source code contains this example in the chapter5/splitter directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=SplitterABCTest
mvn test -Dtest=SpringSplitterABCTest
```

Now you've seen the Splitter in action. To know how to tell Camel what it should split, you need to understand how it works.

How the Splitter works

The Splitter works something like a big iterator that iterates through something and processes each entry. The sequence diagram in figure 5.7 shows more details about how this *big iterator* works.

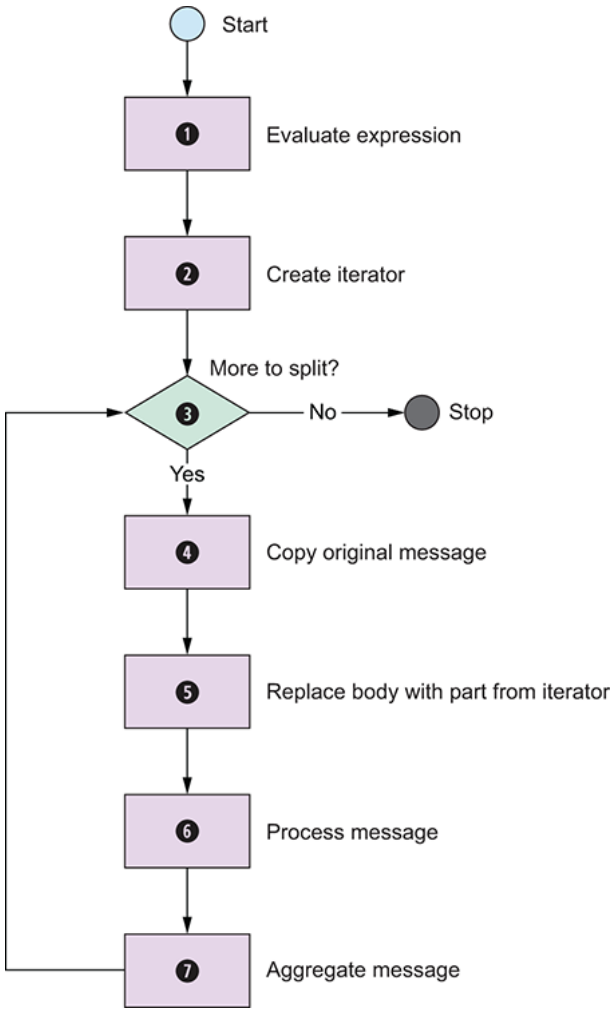

Figure 5.7 A sequence diagram showing how the Splitter works internally, by using an iterator to iterate through the message and process each entry

When working with the Splitter, you have to configure an `Expression`, which is evaluated ❶ when a message arrives. In listing 5.4, the evaluation returned the message body. The result from the evaluation is used to create `java.util.Iterator` ❷.

WHAT CAN BE ITERATED?

When Camel creates the iterator ❷, it supports a range of types. Camel knows how to iterate through the following types: `Collection`, `Iterator`, `Iterable`, `org.w3c.dom`, `NodeList`, `String` (with entries separated by commas) and arrays. Any other type will be iterated once.

Then the Splitter uses the iterator ❸ until there's no more data. Each message to be sent out of the iterator is a copy of the original message ❹, which has had its message body replaced (`org.apache.camel.Message.setBody` is called) with the part from the it-

erator ❺. In listing 5.4, there would be three parts: each of the letters *A*, *B*, and *C*. The message to be sent out is then processed ❻, and when the processing is done, the message may be aggregated ❼ (more about this in section 5.3.4).

The Splitter will decorate each message it sends out with properties on the `Exchange`, which are listed in table 5.8.

Table 5.8 Properties on the `Exchange` related to the Splitter EIP

Property	Type	Description
`Exchange.SPLIT_INDEX`	`int`	The index for the current message being processed. The index is zero-based.
`Exchange.SPLIT_SIZE`	`int`	The total number of messages that the original message has been split into. Note that this information isn't available in streaming mode (see section 5.3.3 for more details about streaming).
`Exchange.SPLIT_COMPLETE`	`boolean`	Whether or not this is the last message being processed.

You may find yourself needing more power to do the splitting, such as to dictate exactly how a message should be split. And what better power is there than Java? By using Java code, you have the ultimate control and can tackle any situation.

5.3.2 Using beans for splitting

Suppose you need to split messages that contain complex payloads. And suppose the message payload is a `Customer` object containing a list of

`Department`s, and you want to split by `Department`, as illustrated in figure 5.8.

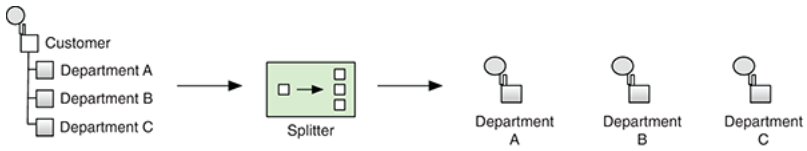

Figure 5.8 Splitting a complex message into submessages by department

The `Customer` object is a simple bean containing the following information (getter and setter methods omitted):

```
public class Customer {
    private int id;
    private String name;
    private List<Department> departments;
}
```

The `Department` object is simple as well:

```
public class Department {
    private int id;
    private String address;
    private String zip;
    private String country;
}
```

You may wonder why you can't split the message as in the previous example, using `split(body())`. The reason is that the message payload (the message body) isn't a `List`, but a `Customer` object. Therefore, you need to tell Camel how to split, which you do as follows:

```
public class CustomerService {
    public List<Department> splitDepartments(Customer customer) {
        return customer.getDepartments();
    }
}
```

The `splitDepartments` method returns a `List` of `Department` objects, which is what you want to split by.

In the Java DSL, you can use the `CustomerService` bean for splitting by telling Camel to invoke the `splitDepartments` method. This is done by using the `method` call expression, as shown in bold:

```
public void configure() throws Exception {
    from("direct:start")
        .split().method(CustomerService.class, "splitDepartments")
            .to("log:split")
            .to("mock:split");
}
```

In XML, you'd have to declare the `CustomerService` in a `<bean>` tag, as follows:

```
<bean id="customerService" class="camelinaction.CustomerService"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <split>
            <method bean="customerService" method="splitDepartments"/>
            <to uri="log:split"/>
            <to uri="mock:split"/>
        </split>
    </route>
</camelContext>
```

The book's source code contains this example in the chapter5/splitter directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=SplitterBeanTest
mvn test -Dtest=SpringSplitterBeanTest
```

The logic in the `splitDepartments` method is simple, but it shows you how to use a method on a bean to do the splitting. In your use cases, you may need more complex logic.

TIP The logic in the `splitDepartments` method seems trivial, and it's possible to use Camel's expression language (Simple) to invoke methods on the message body. In Java DSL, you could define the route as follows: `.split().simple("${body.departments}")`. In XML you'd use the `<simple>` tag instead of the `<method>` tag: `<simple>${body.departments}</simple>`.

The Splitter will usually operate on messages that are loaded into memory. But in some situations, the messages are so big that it's not feasible to have the entire message in memory at once.

5.3.3 Splitting big messages

Rider Auto Parts has an ERP system that contains inventory information from all its suppliers. To keep the inventory updated, each supplier must submit updates to Rider Auto Parts. Some suppliers do this once a day, using good old-fashioned files as a means of transport. Those files could be large, so you have to split those files without loading the entire file into memory.

You can do that by using streams, which allow you to read on demand from a stream of data. This resolves the memory issue, because you can read in a chunk of data, process the data, read in another chunk, process the data, and so on.

Figure 5.9 shows the flow of the application used by Auto Rider Parts to pick up the files from the suppliers and update the inventory.

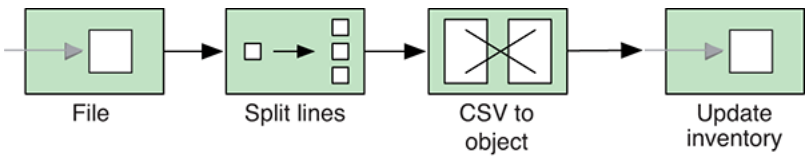

Figure 5.9 A route that picks up incoming files, splits them, and transforms them so they're ready for updating the inventory in the ERP system

We'll revisit this example again in chapter 13, and cover it in much greater detail when we cover concurrency.

Implementing the route outlined in figure 5.9 is easy to do in Camel, as shown in the following listing.

Listing 5.5 Splitting big files by using streaming mode

```
public void configure() throws Exception {
    from("file:target/inventory")
        .log("Starting to process big file: ${header.CamelFileName}")
    .split(body().tokenize("\n")).streaming()    ❷
            .bean(InventoryService.class, "csvToObject")
            .to("direct:update")
    .end()    ❷

        .log("Done processing big file: ${header.CamelFileName}");

    from("direct:update")
        .bean(InventoryService.class, "updateInventory");
}
```

❶ Splits file using streaming mode

❷ Denotes where the splitting route ends

As you can see, all you have to do is enable streaming mode by using `.streaming` ❶. This tells Camel to not load the entire payload into memory, but instead to iterate the payload in a streaming fashion. Also notice the use of `end` ❷ to indicate the end of the splitting route. The `end` in the Java DSL is the equivalent of the end tag `</split>` when using XML.

In XML, you enable streaming by using the `streaming` attribute on the `<split>` tag, as shown in the following listing.

Listing 5.6 Splitting big files by using streaming mode in XML

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file:target/inventory"/>
        <log message="Processing big file: ${header.CamelFileName}"/>
```

```
        <split streaming="true">
            <tokenize token="\n"/>
            <bean beanType="camelinaction.InventoryService"
                  method="csvToObject"/>
            <to uri="direct:update"/>
        </split>
        <log message="Done processing big file: ${header.CamelFileName}"
    </route>
    <route>
        <from uri="direct:update"/>
        <bean beanType="camelinaction.InventoryService"
              method="updateInventory"/>
    </route>
</camelContext>
```

You may have noticed in listings 5.5 and 5.6 that the files are split by using a tokenizer. The *tokenizer* is a powerful feature that works well with streaming. It uses `java.util.Scanner` under the hood, which reads chunks of data into memory. A token must be provided to indicate the boundaries of the chunks. In the preceding code, you use a newline (`\n`) as the token. In this example, the `Scanner` will read the file into memory on a line-by-line basis, resulting in low memory consumption.

NOTE When using streaming mode, be sure the message you're splitting can be split into well-known chunks that can be iterated. You can use the tokenizer (the `java.util.Scanner` used implements `Iterator`) or convert the message body to a type that can be iterated, such as an `Iterator` .

The Splitter EIP in Camel includes an aggregation feature that lets you recombine split messages into single outbound messages, while they're being routed.

5.3.4 Aggregating split messages

Being able to split and aggregate messages again is a powerful mechanism. You could use this to split an order into individual order lines, process them, and then recombine them into a single outgoing message.

This pattern is known as the Composed Message Processor, which we briefly touched on in section 5.1. It's shown in figure 5.1.

The Camel Splitter provides a built-in aggregator, which makes it even easier to aggregate split messages back into single outgoing messages. Figure 5.10 illustrates this principle, with the help of the *ABC* message example.

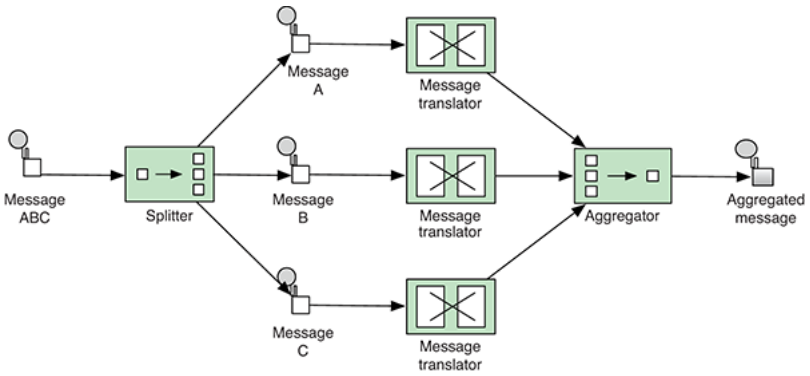

Figure 5.10 The Splitter has a built-in aggregator that can recombine split messages into a combined outgoing message.

Suppose you want to translate each of the *A*, *B*, and *C* messages into a phrase, and have all the phrases combined into a single message again. You can easily do this with the Splitter; all you need to provide is the logic that combines the messages. This logic is created using an `AggregationStrategy` implementation.

Implementing the Camel route outlined in figure 5.10 can be done as follows in the Java DSL. The configuration of the `AggregationStrategy` is shown in bold:

```
from("direct:start")
    .split(body(), new MyAggregationStrategy())
        .log("Split line ${body}")
        .bean(WordTranslateBean.class)
        .to("mock:split")
    .end()
    .log("Aggregated ${body}")
    .to("mock:result");
```

In XML, you have to declare `AggregationStrategy` as a `<bean>` tag, as shown in bold:

```
<bean id="translate" class="camelinaction.WordTranslateBean"/>
<bean id="myAggregationStrategy"
      class="camelinaction.MyAggregationStrategy"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <split strategyRef="myAggregationStrategy">
            <simple>body</simple>
            <log message="Split line ${body}"/>
            <bean ref="translate"/>
            <to uri="mock:split"/>
        </split>
        <log message="Aggregated ${body}"/>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

To combine the split messages back into a single combined message, you use `AggregationStrategy`, as shown in the following listing.

Listing 5.7 Combining split messages back into a single outgoing message

```
public class MyAggregationStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange
        if (oldExchange == null) {
            return newExchange;
        }
        String body = newExchange.getIn().getBody(String.class).trim();
        String existing = oldExchange.getIn().getBody(String.class).trim
        oldExchange.getIn().setBody(existing + "+" + body);
        return oldExchange;
    }
}
```

As you can see, you combine the messages into a single `String` body, with individual phrases (from the message bodies) being separated with `+` signs.

The source code for the book contains this example in the chapter5/splitter directory. You can run it using the following Maven

goals:

```
mvn test -Dtest=SplitterAggregateABCTest
mvn test -Dtest=SpringSplitterAggregateABCTest
```

The example uses the three phrases: *Aggregated Camel rocks*, *Hi mom*, and *Yes it works*. When you run the example, you'll see the console output the aggregated message at the end:

```
INFO  route1 - Split line A
INFO  route1 - Split line B
INFO  route1 - Split line C
INFO  route1 - Aggregated Camel rocks+Hi mom+Yes it works
```

Before we wrap up our coverage of the Splitter, let's take a look at what happens if one of the split messages fails with an exception.

5.3.5 When errors occur during splitting

The Splitter processes messages, and those messages can fail when business logic throws an exception. Camel's error handling is active during the splitting, so the errors you have to deal with in the Splitter are errors that couldn't be handled by the error-handling rules you defined.

You have two choices for handling errors with the Splitter:

- *Stop*—The Splitter will split and process each message in sequence. Suppose the second message failed. In this situation, you could either immediately stop and let the exception propagate back, or you could continue splitting the remainder of the messages, and let the exception propagate back at the end (default behavior).
- *Aggregate*—You could handle the exception in `AggregationStrategy` and decide whether the exception should be propagated back.

Let's look into the choices.

Using stopOnException

The first solution requires you to configure the `stopOnException` option on the Splitter as follows:

```
from("direct:start")
    .split(body(), new MyAggregationStrategy())
        .stopOnException()
        .log("Split line ${body}")
        .bean(WordTranslateBean.class)
        .to("mock:split")
    .end()
    .log("Aggregated ${body}")
    .to("mock:result");
```

In XML, you use the `stopOnException` attribute on the `<split>` tag, as follows:

```
<split strategyRef="myAggregationStrategy" stopOnException="true">
```

The book's source code contains this example in the chapter5/splitter directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=SplitterStopOnExceptionABCTest
mvn test -Dtest=SpringSplitterStopOnExceptionABCTest
```

The second option is to handle exceptions from the split messages in `AggregationStrategy`.

Handling exceptions using AggregationStrategy

`AggregationStrategy` allows you to handle the exception by either ignoring it or letting it be propagated back. Here's how you could ignore the exception.

Listing 5.8 Handling an exception by ignoring it

```
public class MyIgnoreFailureAggregationStrategy
                implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange
        if (newExchange.getException() != null) {
            return oldExchange;  (1)
```

❶

Ignores the exception

```
        }
        if (oldExchange == null) {
            return newExchange;
        }
        String body = newExchange.getIn().getBody(String.class);
        String existing = oldExchange.getIn().getBody(String.class);
        oldExchange.getIn().setBody(existing + "+" + body);
        return oldExchange;
    }
}
```

When handling exceptions in `AggregationStrategy`, you can detect whether an exception occurred by checking the `getException` method from the `newExchange` parameter. The preceding example ignores the exception by returning `oldExchange` ❶.

If you want to propagate back the exception, you need to keep it stored on the aggregated exception, as shown in the following listing.

Listing 5.9 Propagating back an exception

```
public class MyPropagateFailureAggregationStrategy
                implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange
        if (newExchange.getException() != null) {
            if (oldExchange == null) {
                return newExchange;
            } else {
                oldExchange.setException(
          newExchange.getException()); ❶
```

❶

Propagates exception

```
                return oldExchange;
            }
```

```
            }
            if (oldExchange == null) {
                return newExchange;
            }
            String body = newExchange.getIn().getBody(String.class);
            String existing = oldExchange.getIn().getBody(String.class);
            oldExchange.getIn().setBody(existing + "+" + body);
            return oldExchange;
        }
    }
```

As you can see, it requires a bit more work to keep the exception. On the first invocation of the `aggregate` method, the `oldExchange` parameter is `null`, and you return the `newExchange` (which has the exception). Otherwise, you must transfer the exception to `oldExchange` ❶.

WARNING When using a custom `AggregationStrategy` with the Splitter, it's important to know that you're responsible for handling exceptions. If you don't propagate the exception back, the Splitter will assume you've handled the exception and will ignore it.

The book's source code contains this example in the chapter5/splitter directory. You can run it by using the following Maven goals:

```
mvn test -Dtest=SplitterAggregateExceptionABCTest
mvn test -Dtest=SpringSplitterAggregateExceptionABCTest
```

Now you've learned all there is to know about the Splitter. Well, almost all. We'll revisit the Splitter in chapter 13 when we look at concurrency. In the next two sections, you'll see EIPs that support dynamic routing, starting with the Routing Slip pattern.

5.4 The Routing Slip EIP

At times, you need to route messages dynamically. For example, you may have an architecture that requires incoming messages to undergo a sequence of processing steps and business rule validations. Because the steps and validations vary widely, you can implement each step as a sepa-

rate filter. The filter acts as a dynamic model to apply the business rule and validations.

This architecture could be implemented by using the Pipes and Filters EIP together with the Filter EIP. But as often happens with EIPs, there's a better way—in this case, the Routing Slip EIP. The Routing Slip acts as a dynamic router that dictates the next step a message should undergo. Figure 5.11 shows this principle.

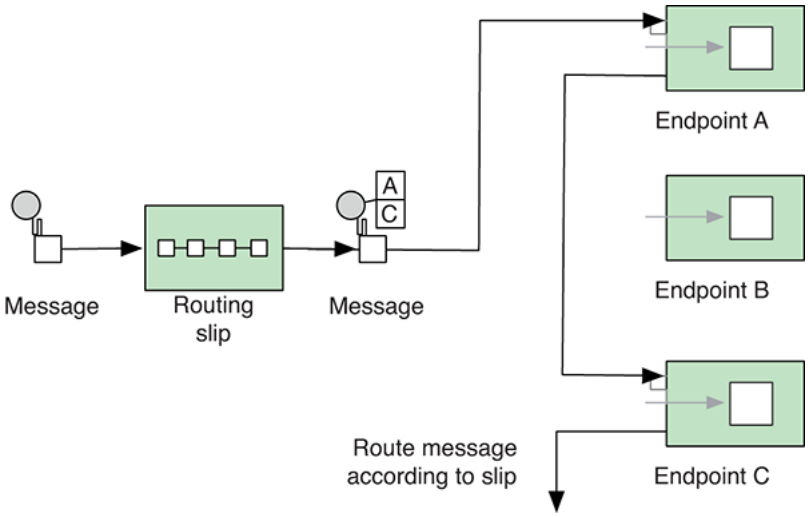

Figure 5.11 The incoming message has a slip attached that specifies the sequence of the processing steps. The Routing Slip EIP reads the slip and routes the message to the next endpoint in the list.

The Camel Routing Slip EIP requires a preexisting header or `Expression` as the attached slip. Either way, the initial slip must be prepared before the message is sent to the Routing Slip EIP.

5.4.1 Using the Routing Slip EIP

We'll start with a simple example that shows how to use the Routing Slip EIP to perform the sequence outlined in figure 5.11.

In the Java DSL, the route is as simple as this:

```
from("direct:start").routingSlip(header("mySlip"));
```

It's also easy in XML:

```
<route>
    <from uri="direct:start"/>
    <routingSlip>
        <header>mySlip</header>
```

```
    </routingSlip>
</route>
```

This example assumes that the incoming message contains the slip in the header with the key `mySlip`. The following test method shows how you should fill out the key:

```
public void testRoutingSlip() throws Exception {
    getMockEndpoint("mock:a").expectedMessageCount(1);
    getMockEndpoint("mock:b").expectedMessageCount(0);
    getMockEndpoint("mock:c").expectedMessageCount(1);
    template.sendBodyAndHeader("direct:start", "Hello World",
                               "mySlip", "mock:a,mock:c");
    assertMockEndpointsSatisfied();
}
```

As you can see, the value of the key is the endpoint URIs separated by commas. The comma is the default delimiter, but the routing slip supports using custom delimiters. For example, to use a semicolon, you could do this:

```
from("direct:start").routingSlip(header("mySlip"), ";");
```

And in XML, you'd do this:

```
<routingSlip uriDelimiter=";">
    <header>mySlip</header>
</routingSlip>
```

This example expects a preexisting header containing the routing slip. But what if the message doesn't contain such a header? In those situations, you have to compute the header in any way you like. In the next example, you'll look at how to compute the header by using a bean.

5.4.2 Using a bean to compute the routing slip header

To keep things simple, the logic to compute a header that contains two or three steps has been kept in a single method, as follows:

```
public class ComputeSlip {
    public String compute(String body) {
        String answer = "mock:a";
        if (body.contains("Cool")) {
            answer += ",mock:b";
        }
        answer += ",mock:c";
        return answer;
    }
}
```

All you have to do now is use this bean to compute the header to be used as the routing slip.

In the Java DSL, you can use the method call expression to invoke the bean and set the header:

```
from("direct:start")
    .setHeader("mySlip").method(ComputeSlip.class)
    .routingSlip(header("mySlip"));
```

In XML, you can do it as follows:

```
<route>
    <from uri="direct:start"/>
    <setHeader headerName="mySlip">
        <method beanType="camelinaction.ComputeSlip"/>
    </setHeader>
    <routingSlip>
        <header>mySlip</header>
    </routingSlip>
</route>
```

In this example, you use a method call expression to set a header that's then used by the routing slip. But you might want to skip the step of setting the header and instead use the expression directly.

5.4.3 Using an Expression as the routing slip

Instead of using a header expression, you can use any other `Expression` to generate the routing slip. For example, you could use a method call ex-

pression, as covered in the previous section. Here's how you'd do so with the Java DSL:

```
from("direct:start")
    .routingSlip(method(ComputeSlip.class));
```

The equivalent XML is as follows:

```
<route>
    <from uri="direct:start"/>
    <routingSlip>
        <method beanType="camelinaction.ComputeSlip"/>
    </routingSlip>
</route>
```

Another way of using the Routing Slip EIP in Camel is to use beans and annotations.

5.4.4 Using @RoutingSlip annotation

The `@RoutingSlip` annotation allows you to turn a regular bean method into the Routing Slip EIP. Let's go over an example.

Suppose you have the following `SlipBean`:

```
public class SlipBean {
    @RoutingSlip
    public String slip(String body) {
        String answer = "mock:a";
        if (body.contains("Cool")) {
            answer += ",mock:b";
        }
        answer += ",mock:c";
        return answer;
    }
}
```

As you can see, all this does is annotate the `slip` method with `@RoutingSlip`. When Camel invokes the `slip` method, it detects the `@RoutingSlip` annotation and continues routing according to the Routing Slip EIP.

WARNING When using `@RoutingSlip`, it's important to not use `routingSlip` in the DSL at the same time. When using both, Camel will double up using the Routing Slip EIP, which isn't the intention. Instead, do as shown in the following example.

Notice that there's no mention of the routing slip in the DSL. The route is just invoking a bean:

```
from("direct:start").bean(SlipBean.class);
```

Here it is in the XML DSL:

```
<bean id="myBean" class="camelinaction.SlipBean"/>

<route>
    <from uri="direct:start"/>
    <bean ref="myBean"/>
</route>
```

Why might you want to use this? Well, by using `@RoutingSlip` on a bean, it becomes more flexible in the sense that the bean is accessible using an endpoint URI. Any Camel client or route could easily send a message to the bean and have it continue being routed as a routing slip.

Messages can also be sent "by hand" to the bean by using `ProducerTemplate`. A template class, in general, is a utility class that simplifies access to an API—in this case, the `Producer` interface. For example, by using `ProducerTemplate`, you could send a message to the bean like this:

```
ProducerTemplate template = ...
template.sendBody("bean:myBean", "Camel rocks");
```

That *Camel rocks* message would then be routed as a routing slip with the slip generated as the result of the `myBean` method invocation.

The source code for the book contains the examples we've covered in the chapter5/routingslip directory. You can try them by using the following

Maven goals:

```
mvn test -Dtest=RoutingSlipSimpleTest
mvn test -Dtest=SpringRoutingSlipSimpleTest
mvn test -Dtest=RoutingSlipHeaderTest
mvn test -Dtest=SpringRoutingSlipHeaderTest
mvn test -Dtest=RoutingSlipTest
mvn test -Dtest=SpringRoutingSlipTest
mvn test -Dtest=RoutingSlipBeanTest
mvn test -Dtest=SpringRoutingSlipBeanTest
```

You've now seen the Routing Slip EIP in action.

5.5 The Dynamic Router EIP

In the previous section, you learned that the Routing Slip pattern acts as a dynamic router. What's the difference between the Routing Slip and Dynamic Router EIPs? The difference is minimal: the Routing Slip needs to compute the slip up front, whereas the Dynamic Router will evaluate on the fly where the message should go next.

5.5.1 Using the Dynamic Router

Just like the Routing Slip, the Dynamic Router requires you to provide *logic*, which determines where the message should be routed. Such logic is easily implemented by using Java code, and in this code you have total freedom to determine where the message should go next. For example, you might query a database or a rules engine to compute where the message should go.

The following listing shows the Java bean used in the example.

Listing 5.10 Java bean deciding where the message should be routed next

```
public class DynamicRouterBean {

    public String route(String body,
        @Header(Exchange.SLIP_ENDPOINT) String previous) {   1
```

❶

Previous endpoint URI

```
        return whereToGo(body, previous);
    }

    private String whereToGo(String body, String previous) {
        if (previous == null) {
            return "mock://a";
        } else if ("mock://a".equals(previous)) {
            return "language://simple:Bye ${body}";
        } else {
      return null;  ❷
```

❷

Ends Dynamic Router

```
        }
    }
}
```

The idea with the Dynamic Router is to let Camel keep invoking the `route` method until it returns `null`. The first time the `route` method is invoked, the `previous` parameter will be `null` ❶. On every subsequent invocation, the `previous` parameter contains the endpoint URI of the last step.

As you can see in the `whereToGo` method, you use this fact and return different URIs depending on the previous step. When the dynamic router is to end, you return `null` ❷. It's very important that the Dynamic Router must return `null` at some point—otherwise, Camel will route the message forever.

Using the Dynamic Router from the Java DSL is easy to do:

```
from("direct:start")
    .dynamicRouter(method(DynamicRouterBean.class, "route"))
    .to("mock:result");
```

The same route in XML is just as easy, as shown here:

```
<bean id="myDynamicRouter" class="camelinaction.DynamicRouterBean"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <dynamicRouter>
            <method ref="myDynamicRouter" method="route"/>
        </dynamicRouter>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

The book's source code contains this example in the chapter5/dynamicrouter directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=DynamicRouterTest
mvn test -Dtest=SpringDynamicRouterTest
```

You can also use the Dynamic Router annotation.

5.5.2 Using the @DynamicRouter annotation

To demonstrate how to use the `@DynamicRouter` annotation, let's change the previous example to use the annotation instead. To do that, annotate the Java code from listing 5.10 as follows:

```
@DynamicRouter
public String route(String body,
                    @Header(Exchange.SLIP_ENDPOINT) String previous
    ...
}
```

The next step is to invoke the `route` method on the bean, as if it were a regular bean. That means you shouldn't use the Routing Slip EIP in the route, but use a bean instead.

In the Java DSL, this is done as follows:

```
from("direct:start")
    .bean(DynamicRouterBean.class, "route")
    .to("mock:result");
```

In XML, you likewise change the `<dynamicRouter>` to a `<bean>` tag:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <bean ref="myDynamicRouter" method="route"/>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

WARNING When using `@DynamicRouter`, it's important to not use `dynamicRouter` in the DSL at the same time. Instead, do as shown in the preceding example.

The book's source code contains this example in the chapter5/dynamicrouter directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=DynamicRouterAnnotationTest
mvn test -Dtest=SpringDynamicRouterAnnotationTest
```

This concludes the coverage of the dynamic routing patterns. In the next section, you'll learn about Camel's built-in Load Balancer EIP, which is useful when an existing load-balancing solution isn't in place.

5.6 The Load Balancer EIP

You may already be familiar with the load-balancing concept in computing. *Load balancing* is a technique to distribute workload across computers or other resources "in order to get optimal resource utilization, maximize throughput, minimize response time, and avoid overload" (http://en.wikipedia.org/wiki/Load_balancer). This service can be provided either in the form of a hardware device or as a piece of software, such as the Load Balancer EIP in Camel.

NOTE The Load Balancer wasn't covered in the EIP book, but will likely be added if there's a second edition of the book.

In this section, we introduce the Load Balancer EIP by walking through an example. Then, in section 5.6.2, we present the various types of load balancers Camel offers out of the box. We focus on the failover type in section 5.6.3 and finally show how to build your own load balancer in section 5.6.4.

5.6.1 Introducing the Load Balancer EIP

The Camel Load Balancer EIP is a processor that implements the `org.apache.camel. processor.loadbalancer.LoadBalancer` interface. `LoadBalancer` offers methods to add and remove processors that should participate in the load balancing.

By using processors instead of endpoints, the load balancer is capable of balancing anything you can define in your Camel routes. But, that said, you'll most often balance across a number of remote services. Such an example is illustrated in figure 5.12, where a Camel application needs to load balance across two services.

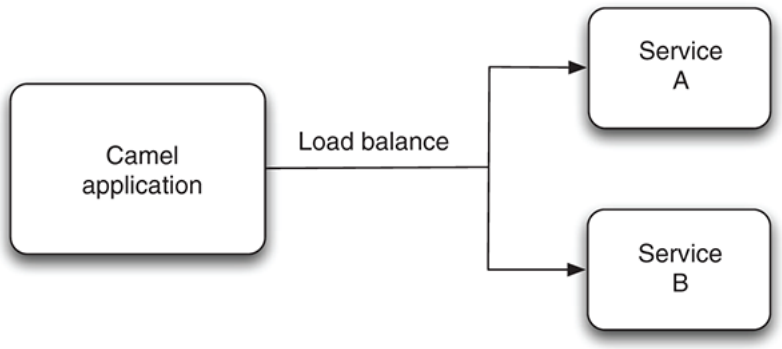

Figure 5.12 A Camel application load-balances across two services.

When using the Load Balancer EIP, you have to select a balancing strategy. A common and understandable strategy is to take turns among the services; this is known as the *round-robin strategy*. In section 5.6.2, we'll take a look at all the strategies Camel provides out of the box.

Let's look at how to use the Load Balancer with the round-robin strategy. Here's the Java DSL with the Load Balancer:

```
from("direct:start")
    .loadBalance().roundRobin()
        .to("seda:a").to("seda:b")
    .end();
from("seda:a")
    .log("A received: ${body}")
    .to("mock:a");
from("seda:b")
    .log("B received: ${body}")
    .to("mock:b");
```

The equivalent route in XML is as follows:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <roundRobin/>
        <to uri="seda:a"/>
        <to uri="seda:b"/>
    </loadBalance>
</route>
<route>
    <from uri="seda:a"/>
    <log message="A received: ${body}"/>
    <to uri="mock:a"/>
</route>
<route>
    <from uri="seda:b"/>
    <log message="B received: ${body}"/>
    <to uri="mock:b"/>
</route>
```

In this example, you use the SEDA component to simulate the remote services. In a real-life situation, the remote services could be a RESTful web service.

Suppose you start sending messages to the route. The first message would be sent to the `seda:a` endpoint, and the next would go to `seda:b`. The third message would start over and be sent to `seda:a`, and so forth.

The book's source code contains this example in the chapter5/loadbalancer directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=LoadBalancerTest
mvn test -Dtest=SpringLoadBalancerTest
```

If you run the example, the console will output something like this:

```
[Camel Thread 0 - seda://a] INFO route2 - A received: Hello
[Camel Thread 1 - seda://b] INFO route3 - B received: Camel rocks
[Camel Thread 0 - seda://a] INFO route2 - A received: Cool
[Camel Thread 1 - seda://b] INFO route3 - B received: Bye
```

The next section reviews the various load-balancing strategies you can use with the Load Balancer EIP.

5.6.2 Using load-balancing strategies

A load-balancing strategy dictates which processor should process an incoming message—it's up to each strategy how the processor is chosen. Camel allows the six strategies listed in table 5.9.

Table 5.9 Load-balancing strategies provided by Camel

Strategy	Description
Random	Chooses a processor randomly.
Round-robin	Chooses a processor in a round-robin fashion, which spreads the load evenly. This is a classic and well-known strategy. We covered this in section 5.6.1.
Sticky	Uses an expression to calculate a correlation key that dictates the processor chosen. You can think of this as the session ID used in HTTP requests.
Topic	Sends the message to all processors. This is like sending to a JMS topic.
Failover	Retries using another processor. We cover this in section 5.6.3.
Custom	Uses your own custom strategy. This is covered in section 5.6.4.

The first four strategies in table 5.9 are easy to set up and use in Camel. For example, using the random strategy is just a matter of specifying it in the Java DSL:

```
from("direct:start")
    .loadBalance().random()
        .to("seda:a").to("seda:b")
    .end();
```

It's similar in XML:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <random/>
        <to uri="seda:a"/>
```

```
        <to uri="seda:b"/>
    </loadBalance>
</route>
```

The sticky strategy requires that you provide a correlation expression, which is used to calculate a hashed value to indicate which processor should be used. Suppose your messages contain a header indicating different levels. By using the sticky strategy, you can have messages with the same level choose the same processor over and over again.

In the Java DSL, you'd provide the expression by using a header expression, as shown here:

```
from("direct:start")
    .loadBalance().sticky(header("type"))
        .to("seda:a").to("seda:b")
    .end();
```

In XML, you'd do the following:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <sticky>
            <correlationExpression>
                <header>type</header>
            </correlationExpression>
        </sticky>
        <to uri="seda:a"/>
        <to uri="seda:b"/>
    </loadBalance>
</route>
```

The book's source code contains examples of using the strategies listed in table 5.9 in the chapter5/loadbalancer directory. To try the random, sticky, circuit breaker, or topic strategies, use the following Maven goals:

```
mvn test -Dtest=RandomLoadBalancerTest
mvn test -Dtest=SpringRandomLoadBalancerTest
mvn test -Dtest=StickyLoadBalancerTest
mvn test -Dtest=SpringStickyLoadBalancerTest
```

```
mvn test -Dtest=CircuitBreakerLoadBalancerTest
mvn test -Dtest=SpringCircuitBreakerLoadBalancerTest
mvn test -Dtest=TopicLoadBalancerTest
mvn test -Dtest=SpringTopicLoadBalancerTest
```

The failover strategy is a more elaborate strategy, which we cover next.

5.6.3 Using the failover load balancer

Load balancing is often used to implement *failover*—the continuation of a service after a failure. The Camel failover load balancer detects the failure when an exception occurs and reacts by letting the next processor take over processing the message.

Given the following route snippet, the failover will always start by sending the messages to the first processor (`direct:a`) and only in the case of a failure will it let the next processor (`direct:b`) take over:

```
from("direct:start")
    .loadBalance().failover()
        .to("direct:a").to("direct:b")
    .end();
```

The equivalent snippet in XML is as follows:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <failover/>
        <to uri="direct:a"/>
        <to uri="direct:b"/>
    </loadBalance>
</route>
```

The book's source code contains this example in the chapter5/loadbalancer directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=FailoverLoadBalancerTest
mvn test -Dtest=SpringFailoverLoadBalancerTest
```

If you run the example, it will send in four messages. The second message will fail over and be processed by the `direct:b` processor. The other three messages will be processed successfully by `direct:a`.

In this example, the failover load balancer will react to any kind of exception being thrown, but you can provide it with a number of exceptions to react to.

Suppose you want to fail over only if an `IOException` is thrown (which indicates communication errors with remote services, such as no connection). This is easy to configure, as shown in the Java DSL:

```
from("direct:start")
    .loadBalance().failover(IOException.class)
        .to("direct:a").to("direct:b")
     .end();
```

Here it's configured in XML:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <failover>
            <exception>java.io.IOException</exception>
        </failover>
        <to uri="direct:a"/>
        <to uri="direct:b"/>
    </loadBalance>
</route>
```

In this example, only one exception is specified, but you can specify multiple exceptions, as follows:

```
from("direct:start")
    .loadBalance().failover(IOException.class, SQLException.class)
        .to("direct:a").to("direct:b")
     .end();
```

In XML, you do as follows:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <failover>
            <exception>java.io.IOException</exception>
            <exception>java.sql.SQLException</exception>
        </failover>
        <to uri="direct:a"/>
        <to uri="direct:b"/>
    </loadBalance>
</route>
```

You may have noticed in the failover examples that it always chooses the first processor and sends the failover to subsequent processors. You can think of this as the first processor being the master and the others slaves. But the failover load balancer also offers a strategy that combines round-robin with failure support.

Using failover with round-robin

The Camel failover load balancer in round-robin mode gives you the best of both worlds: it distributes the load evenly between the services, and it provides automatic failover.

In this scenario, you have three configuration options on the load balancer to dictate how it operates, as listed in table 5.10.

Table 5.10 Failover load-balancer configuration options

Configuration option	Default	Description
`maximumFailoverAttempts`	`-1`	Specifies the number of failover attempts to try before exhausting (giving up): • Use –1 to attempt forever (never give up). • Use 0 to never fail over (give up immediately). • Use a positive value to specify a number of attempts. For example, a value of 3 will try up to three failover attempts before giving up.
`inheritErrorHandler`	`true`	Specifies whether Camel error handling is being used. When enabled, the load balancer will let the error handler be involved. If disabled, the load balancer will fail over immediately if an exception is thrown.
`roundRobin`	`false`	Specifies whether the load balancer operates in round-robin mode.

To better understand the options in table 5.10 and how the round-robin mode works, let's start with a fairly simple example. In the Java DSL, you have to configure failover with all the options in bold:

```
from("direct:start")
    .loadBalance().failover(1, false, true)
        .to("direct:a").to("direct:b")
    .end();
```

In this example, the `maximumFailoverAttempts` option is set to `1`, which means it will at most try to fail over once (it will make one attempt for the initial request and one more for the failover attempt). If both attempts fail, Camel will propagate the exception back to the caller. The second parameter is set to `false`, which means it isn't inheriting Camel's error handling. This allows the failover load balancer to fail over immediately when an exception occurs, instead of having to wait for the Camel error handler to give up first. The last parameter indicates that it's using the round-robin mode.

In XML, you configure the options as attributes on the `failover` tag:

```
<route>
    <from uri="direct:start"/>
    <loadBalance inheritErrorHandler="false">
        <failover roundRobin="true" maximumFailoverAttempts="1"/>
        <to uri="direct:a"/>
        <to uri="direct:b"/>
    </loadBalance>
</route>
```

The book's source code contains this example in the chapter5/loadbalancer directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=FailoverRoundRobinLoadBalancerTest
mvn test -Dtest=SpringFailoverRoundRobinLoadBalancerTest
```

If you're curious about the `inheritErrorHandler` configuration option, take a look at the following examples in the book's source code:

```
mvn test -Dtest=FailoverInheritErrorHandlerLoadBalancerTest
mvn test -Dtest=SpringFailoverInheritErrorHandlerLoadBalancerTest
```

This concludes our tour of the failover load balancer. The next section explains how to implement and use your own custom strategy, which you may want to do when you need to use special load-balancing logic.

5.6.4 Using a custom load balancer

Custom load balancers allow you to be in full control of the balancing strategy in use. For example, you could build a strategy that acquires load statistics from various services and picks the service with the lowest load.

Let's look at an example. Suppose you want to implement a priority-based strategy that sends gold messages to a certain processor and the remainder to a secondary destination. Figure 5.13 illustrates this principle.

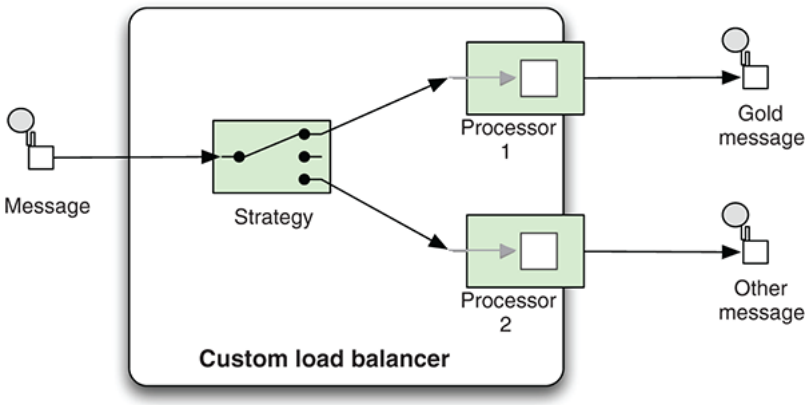

Figure 5.13 Using a custom load balancer to route gold messages to processor 1 and other messages to processor 2

When implementing a custom load balancer, you'll often extend the `SimpleLoadBalancerSupport` class, which provides a good starting point. The following listing shows how to implement a custom load balancer.

Listing 5.11 Custom load balancer

```
public class MyCustomLoadBalancer extends SimpleLoadBalancerSupport {
    public boolean process(Exchange exchange) throws Exception {
        Processor target = chooseProcessor(exchange);
        target.process(exchange);
    }
    @Override
    protected Processor chooseProcessor(Exchange exchange) {
        String type = exchange.getIn().getHeader("type", String.class);
        if ("gold".equals(type)) {
            return getProcessors().get(0);    (1)
```

selects processor 1

```
        } else {
            return getProcessors().get(1);
```

selects processor 2

```
        }
    }
}
```

As you can see, it doesn't take much code. In the `process` method, you invoke the `chooseProcessor` method, which is the strategy that picks the processor to process the message. In this example, it'll pick the first processor if the message is a gold type, and the second processor if not.

In the Java DSL, you use a custom load balancer as shown in bold:

```
from("direct:start")
    .loadBalance(new MyCustomLoadBalancer())
        .to("seda:a").to("seda:b")
    .end();
```

In XML, you need to declare a `<bean>` tag:

```
<bean id="myCustom" class="camelinaction.MyCustomLoadBalancer"/>
```

You then refer to that bean from the `<custom>` tag inside the `<loadBalance>` tag:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <custom ref="myCustom"/>
        <to uri="seda:a"/>
        <to uri="seda:b"/>
```

```
    </loadBalance>
  </route>
```

The book's source code contains this example in the chapter5/loadbalancer directory. You can try it by using the following Maven goals:

```
mvn test -Dtest=CustomLoadBalancerTest
mvn test -Dtest=SpringCustomLoadBalancerTest
```

3
We've now covered the Load Balancer EIP in Camel, which brings us to the end of our long journey to visit five great EIPs implemented in Camel.

5.7 Summary and best practices

Since the arrival of the *Enterprise Integration Patterns* book on the scene, we've had a common vocabulary, graphical notation, and concepts for designing applications to tackle today's integration challenges. You've encountered these EIPs throughout this book. In chapter 2 we reviewed the most common patterns, and this chapter reviews five of the most complex and sophisticated patterns in great detail. You may view the EIP book as the theory, and Camel as the software implementation of the book.

Here are some EIP best practices to take away from this chapter:

- *Learn the patterns*—Take the time to study the EIPs, especially the common patterns covered in chapter 2 and those presented in this chapter. Consider getting the EIP book to read more about the patterns; great advice is given in the book. The EIP book authors also maintain a website, which is a nice reference for the patterns. Lately they've even updated it to include concrete examples using Camel: www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.ht The patterns are universal, and the knowledge you gain when using EIPs with Camel is something you can take with you.
- *Use the patterns*—If you have a problem you don't know how to resolve, there's a good chance others have scratched that itch before. Consult the EIP book and the online Camel patterns catalog: http://camel.apache.org/eip. Another highly recommended EIP pattern

book is *Camel Design Patterns* by Bilgin Ibryam (Leanpub, 2016, https://leanpub.com/camel-design-patterns).

- *Start simply*—When learning to use an EIP, you should create a simple test to try the pattern and learn how to use it. Having too many new moving parts in a Camel route can clutter your view and make it difficult to understand what's happening and, maybe, why it doesn't do what you expect.
- *Come back to this chapter*—If you're going to use any of the five EIPs covered in this chapter, we recommend you reread the relevant parts of the chapter. These patterns are sophisticated and have many features and options to tweak.

The next chapter covers the use of components with Camel. You've already used components, such as the file and SEDA components. But there's much more to components, so we devote an entire chapter to cover them in detail.