12

Transactions and idempotency

This chapter covers

- Understanding why you need transactions
- Using and configuring transactions
- Understanding the differences between local and global transactions
- Using transactions with messaging and databases
- Rolling back transactions
- Compensating when transactions aren't supported
- Preventing duplicate messages by using idempotency
- Learning about the idempotent repository implementations shipped out of the box

To help explain transactions, let's look at an example from real life. You may well have ordered this book from Manning's online bookstore, and if you did, you likely followed these steps:

1. Find the book *Camel in Action*, 2nd Edition.
2. Put the book into the basket.
3. Maybe continue shopping and look for other books.
4. Go to the checkout.
5. Enter shipping and credit card details.
6. Confirm the purchase.
7. Wait for the confirmation.
8. Leave the web store.

What seems like an everyday scenario is a fairly complex series of events. You have to put books in the basket before you can check out; you must fill in the shipping and credit card details before you can confirm the purchase; if your credit card is declined, the purchase won't be confirmed;

and so on. The ultimate resolution of this transaction is one of two states: either the purchase is accepted and confirmed, or the purchase is declined, leaving your credit card balance uncharged.

This particular story involves computer systems because it's about using an online bookstore, but the same main points happen when you shop in the supermarket. Either you leave the supermarket with your groceries or without.

In the software world, transactions are often explained in the context of SQL statements manipulating database tables—updating or inserting data. While the transaction is in progress, a system failure could occur, and that would leave the transaction's participants in an inconsistent state. That's why the series of events is described as *atomic*: either they all are completed or they all fail—it's all or nothing. In transactional terms, they either *commit* or *roll back*.

NOTE You probably know about the database ACID properties, so we won't explain what *atomic*, *consistent*, *isolated*, and *durable* mean in the context of transactions. If you aren't familiar with ACID, the Wikipedia page is a good place to start learning about it: http://en.wikipedia.org/wiki/ACID.

In this chapter, we'll first look at the reasons why you should use transactions (in the context of Rider Auto Parts). Then we'll look at transactions in more detail and at Spring's transaction management, which orchestrates the transactions. You'll learn about the difference between local and global transactions and how to configure and use transactions. In the middle of the chapter, you'll see how to compensate when you're using resources that don't support transactions. The last part of the chapter covers *idempotency*, which deals with handling duplicate messages. This can often happen when transactions can't be used, and data exchange between distributed systems may have to replay messages to deal with an unreliable network between the systems. The last section of the chapter covers a common use-case of clustering an application and using clustered idempotency to coordinate work between the nodes, so each node doesn't process duplicate messages.

12.1 Why use transactions?

Using transactions makes sense for many reasons. But before focusing on using transactions with Camel, let's look at what can go wrong when you *don't* use transactions.

In this section, you'll review an application that Rider Auto Parts uses to collect metrics that will be published to an incident management system. You'll see what goes wrong when the application doesn't use transactions, and then you'll apply transactions to the application.

12.1.1 The Rider Auto Parts partner integration application

Lately, Rider Auto Parts has had a dispute with a partner about whether its service meets the terms of the service-level agreement (SLA). When such incidents occur, it's often a labor-intensive task to investigate and remedy the incident.

In light of this, Rider Auto Parts has developed an application to record what happens, as evidence for when a dispute comes up. The application periodically measures the communication between Rider Auto Parts and its external partner servers. The application records performance and up-time metrics, which are sent to a JMS queue, where the data awaits further processing.

Rider Auto Parts already has an existing incident management application with a web-user interface for upper management. What's missing is an application to populate the collected metrics to the database used by the incident management application. Figure 12.1 illustrates the scenario.

Figure 12.1 Partner reports are received from the JMS broker, transformed in Camel to SQL format, and then written to the database.

It's a fairly simple task: a JMS consumer listens for new messages on the JMS queue ❶. Then the data is transformed from XML to SQL ❷ before it's written to the database ❸.

In no time, you can come up with a route that matches figure 12.1:

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

```
    <propertyPlaceholder id="properties"
                         location="camelinaction/sql.properties"/>

    <route id="partnerToDB">
      <from uri="activemq:queue:partners"/>
      <bean ref="partner" method="toMap"/>
      <to uri="sql:{{sql-insert}}?dataSource=#myDataSource"/>
    </route>
  </camelContext>
```

The reports are sent to the JMS queue in a simple in-house XML format, like this:

```
<?xml version="1.0"?>
<partner id="123">
    <date>201702250815</date>
    <code>200</code>
    <time>3921</time>
</partner>
```

The database table that stores the data is also mapped easily because it has the following layout:

```
create table partner_metric
    ( partner_id varchar(10), time_occurred varchar(20),
      status_code varchar(3), perf_time varchar(10) )
```

That leaves you with the fairly simple task of mapping the XML to the database. Because you're pragmatic and want to make a simple and elegant solution that anybody should be capable of maintaining in the future, you decide not to bring in the big guns with the Java Persistence API (JPA) or Hibernate. You put the mapping code shown in the following listing in a good old-fashioned bean.

Listing 12.1 Using a bean to map from XML to SQL

```
import org.apache.camel.language.XPath;

public class PartnerServiceBean {
```

```
  public Map toMap(@XPath("partner/@id") int id,    ❶
                   @XPath("partner/date/text()") String date,
                   @XPath("partner/code/text()") int statusCode,
                   @XPath("partner/time/text()") long responseTime)
    Map map = new HashMap();    ❷
```

❶ Extracts data from XML payload

❷ Constructs Map with values to insert using SQL

```
    map.put("id", id);
    map.put("date", date);
    map.put("code, statusCode);
    map.put("time", responseTime);
    return map;
  }
}
```

Coding the mapping logic that extracts the data from XML to a `Map` in these 10 or so lines was faster than getting started on the JPA wagon or opening any heavyweight and proprietary mapping software.

First, you define the method to accept the four values to be mapped. Notice that you use the `@XPath` annotation to grab the data from the XML document ❶. Then you use `Map` to store the input values ❷. The SQL `INSERT` statement is externalized from Java code and defined in a properties file, which you name `sql.properties`:

```
sql-insert=insert into partner_metric (partner_id, time_occurred,
status_code, perf_time) values (:#id, :#date, :#code, :#time)
```

Notice that the keys from the `Map` ❷ match the values in the SQL statement (for example, `:#id` matching the ID, and `:#date` matching the

date). Using `:#key` is how the Camel SQL component supports mapping from the `Message` body or headers to the SQL dynamic placeholders.

NOTE In the first edition of this book, we used the Camel JDBC component for this example. This time we're using use the SQL component, because it's been improved. For example, the SQL component uses prepared statements, whereas JDBC uses regular statements; this should yield better performance on the database. We, the authors, also admire the camel-elsql component that allows you to use a lightweight DSL to define the SQL queries.

To test this, you can crank up a unit test as follows:

```
public void testSendPartnerReportIntoDatabase() throws Exception {
    String sql = "select count(*) from partner_metric";
    assertEquals(0, jdbc.queryForInt(sql));    ❶
```

---

❶ Asserts there are no rows in database

```
    String xml = "<?xml version=\"1.0\"?>
                + <partner id=\"123\"><date>201702250815</date>
                + <code>200</code><time>4387</time></partner>";
    template.sendBody("activemq:queue:partners", xml);
    Thread.sleep(5000);
    assertEquals(1, jdbc.queryForInt(sql));    ❷
```

---

❷ Asserts one row was inserted into database

```
}
```

This test method outlines the principle. First you check that the database is empty ❶. Then you construct sample XML data and send it to the JMS queue using the Camel `ProducerTemplate`. Because the processing of the

JMS message is asynchronous, you must wait a bit to let it process (the book's source code uses `NotifyBuilder` to wait instead of `Thread.sleep`). At the end, you check that the database contains one row ❷.

12.1.2 Setting up the JMS broker and the database

To run this test, you need to use a local JMS broker and a database. You can use Apache ActiveMQ as the JMS broker and Apache Derby as the database. Derby can be used as an in-memory database without the need to run it separately. ActiveMQ is an extremely versatile broker, and it's even embeddable in unit tests.

All you have to do is master a bit of Spring XML magic to set up the JMS broker and the database, as shown in the following listing.

Listing 12.2 XML configuration for the Camel route, JMS broker, and database

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

<bean id="partner" class="camelinaction.PartnerServiceBean"/>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <propertyPlaceholder id="properties"
                           location="camelinaction/sql.properties"/>
  <route id="partnerToDB">
    <from uri="activemq:queue:partners"/>
    <bean ref="partner" method="toMap"/>
    <to uri="sql:{{sql-insert}}?dataSource=#myDataSource"/>
  </route>
</camelContext>
```

```
<bean id="activemq"     ❶
```

❶

Configures ActiveMQ component

```
        class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>

<broker:broker useJmx="false" persistent="false" brokerName="localhost">
 <broker:transportConnectors>     ❷
```

❷

Sets up embedded JMS broker

```
    <broker:transportConnector uri="tcp://localhost:61616"/>
  </broker:transportConnectors>
</broker:broker>

<bean id="myDataSource"
 class="org.springframework.jdbc.datasource.DriverManagerDataSource">     ❸
```

❸

Sets up database

```
  <property name="driverClassName"
            value="org.apache.derby.jdbc.EmbeddedXADataSource"/>
  <property name="url" value="jdbc:derby:memory:order;create=true"/>
</bean>

</beans>
```

In listing 12.2, you first define the partner bean from listing 12.1 as a Spring bean that you'll use in the route. Then, to allow Camel to connect to ActiveMQ, you must define it as a Camel component ❶. The `brokerURL`

property is configured with the URL for the remote ActiveMQ broker, which, in this example, happens to be running on the same machine. Then you set up a local embedded ActiveMQ broker ❷, which is configured to use TCP connectors. Finally, you set up the JDBC data source ❸.

USING VM INSTEAD OF TCP WITH AN EMBEDDED ACTIVEMQ BROKER

If you use an embedded ActiveMQ broker, you can use the VM protocol instead of TCP; doing so bypasses the entire TCP stack and is much faster. For example, in listing 12.2, you could use `vm://localhost` instead of `tcp://localhost:61616`. The `localhost` in `vm://localhost` is the broker name, not a network address. For example, you could use `vm://myCoolBroker` as the broker name and configure the name on the broker tag accordingly: `brokerName="myCoolBroker"`.

A plausible reason that you're using `vm://localhost` in listing 12.2 is that the engineers are lazy, and they changed the protocol from TCP to VM but left the broker name as `localhost`.

Embedding an ActiveMQ broker is a good use-case for unit and integration tests, and for some application servers that provide messaging out of the box such as Apache ServiceMix and JBoss Fuse. For other use-cases, it's recommended to run ActiveMQ standalone, which allows you to separate the brokers from your applications. ServiceMix and JBoss Fuse users may consider an architecture in which they configure some nodes to be dedicated ActiveMQ brokers and other nodes to host their applications. This allows you to cleanly separate brokers from applications, which is a recommended practice. But the one-size-fits-all rule applies here; in some use-cases, colocating the ActiveMQ brokers with your applications can be good practice (for example, if the messaging load is light, and you don't need to scale the brokers independently from your applications).

The full source code for this example is located in the chapter12/riderautoparts-partner directory, and you can try out the example by running the following Maven goal:

```
mvn test -Dtest=RiderAutoPartsPartnerTest
```

In the source code, you'll also see how we prepared the database by creating the table and dropping it after testing.

12.1.3 The story of the lost message

The previous test is testing a positive situation, but what happens if the connection to the database fails? How can you test that?

Chapter 9 covered how to simulate a connection failure using Camel interceptors. Writing a unit test is just a matter of putting all that logic in a single method, as shown in the following listing.

Listing 12.3 Simulating a connection failure that causes lost messages

```
public void testNoConnectionToDatabase() throws Exception {
    NotifyBuilder notify = new NotifyBuilder(context).whenDone(1).create

    RouteBuilder rb = new RouteBuilder() {    ❶
```

❶ Simulates no connection to database

```
        public void configure() throws Exception {
            interceptSendToEndpoint("sql:*")
                .throwException(new ConnectException("Cannot connect"));
        }
    };

    RouteDefinition route = context.getRouteDefinition("partnerToDB");
    route.adviceWith(context, rb);    ❷
```

❷ Advises simulation into existing route

```
    String sql = "select count(*) from partner_metric";
```

```
    assertEquals(0, jdbc.queryForInt(sql));  ❸
```

❸

SQL to select number of rows in the database

```
    String xml = "<?xml version=\"1.0\"?>
                + <partner id=\"123\"><date>201611150815</date>
                + <code>200</code><time>4387</time></partner>";

    template.sendBody("activemq:queue:partners", xml);

    assertTrue(notify.matches(10, TimeUnit.SECONDS));

    assertEquals(0, jdbc.queryForInt(sql));  ❹
```

❹

Asserts no rows inserted into database

```
}
```

To test a failed connection to the database, you need to intercept the routing to the database and simulate the error. You do this with `RouteBuilder`, where you define this scenario ❶. Next you need to add the interceptor with the existing route ❷, which is done using the `adviceWith` method. The remainder of the code is almost identical to the previous test, but you test that no rows are added to the database, as the `select count(*)` ❸ SQL query should return `0` rows ❹.

NOTE You can read about simulating errors by using interceptors in chapter 9, section 9.4.3.

The test runs successfully. But what happened to the message you sent to the JMS queue? It wasn't stored in the database, so where did it go?

It turns out that the message is lost because you're not using transactions. By default, the JMS consumer uses *auto-acknowledge mode*: the client acknowledges the message when it's received, and the message is dequeued from the JMS broker.

What you must do instead is use *transacted-acknowledge mode*. We'll look at how to do this in section 12.3, but first we'll discuss how transactions work in Camel.

NOTE The JMS specification also defines a *client-acknowledge mode*. This mode isn't often used with Camel, as you (the client) need to call the `acknowledge` method on the JMS message to mark the message as successfully consumed; performing this action requires you to write Java code. The book's source code contains an example of using JMS client-acknowledge mode in the chapter12/riderautoparts-partner directory. You can try this example using the following Maven goal:

```
mvn test -Dtest=RiderAutoPartsPartnerClientAcknowledgeModeTest
```

12.2 Transaction basics

A *transaction* is a series of events. The start of a transaction is often named `begin`, and the end is `commit` (or `rollback` if the transaction isn't successfully completed). Figure 12.2 illustrates this.

Figure 12.2 A transaction is a series of events between `begin` and `commit`.

To demonstrate the sequence in figure 12.2, you could write *locally managed transactions* (the transaction is managed manually in the code). The following code illustrates this:

```
TransactionManager tm = ...
Transaction tx = tm.getTransaction();
try {
    tx.begin();
    ...
    tx.commit();
} catch (Exception e) {
```

```
        tx.rollback();
    }
```

You start the transaction using the `begin` method. Then you have a series of events to do whatever work needs to be done. At the end, you either commit or roll back the transaction, depending on whether an exception is thrown.

You may already be familiar with this principle, and transactions in Camel use the same principle at a higher level of abstraction. In Camel transactions, you don't invoke `begin` and `commit` methods from Java code; you use declarative transactions, which can be configured using Java code or in XML files. Camel doesn't reinvent the wheel and implement a transaction manager, which is a complicated piece of technology to build. Instead, Camel uses Spring's transaction support.

NOTE For more information on Spring's transaction management, see chapter 10, "Transaction Management," in the *Spring Framework Reference Documentation:*http://docs.spring.io/spring/docs/current/spring-framework-reference/html/transaction.html.

Now that we've established that Camel works with Spring's transaction support, let's look at how they work together.

12.2.1 Understanding Spring's transaction support

To understand how Camel works together with Spring's transaction support, take a look at figure 12.3. This figure shows that Spring orchestrates the transaction while Camel takes care of the rest.

SPRING TRANSACTIONMANAGER API VS. IMPLEMENTATIONS

Camel uses Spring Transaction to manage transactions via its `TransactionManager` API. Depending on the kinds of resources that are taking part in the transaction, an appropriate implementation of the transaction manager must be chosen. Spring offers a number of transaction managers out of the box that work for various local transactions such as JMS and JDBC. But for global transactions, you must use a third-party JTA transaction manager implementation; JTA transaction manager is provided by Java EE application servers. Spring doesn't offer that out of the box, only the necessary API abstract that Camel uses. We cover using third-party transaction managers in section 12.3.2.

Figure 12.3 Spring's `TransactionManager` orchestrates the transaction by issuing `begin`s and `commit`s. The entire Camel route is transacted, and the transaction is handled by Spring.

Figure 12.4 adds more details, to illustrate that the JMS broker also plays an active part in the transaction. You can see how the JMS broker, Camel, and the Spring `JmsTransactionManager` work together. `JmsTransactionManager` orchestrates the resources that participate in the transaction ❶, which in this example is the JMS broker.

Figure 12.4 The Spring `JmsTransactionManager` orchestrates the transaction with the JMS broker. The Camel route completes successfully and signals the commit to the `JmsTransactionManager`.

When a message has been consumed from the JMS queue and fed into the Camel application, Camel issues a `begin` ❷ to `JmsTransactionManager`. Depending on whether the Camel route completes with success or failure ❸, `JmsTransactionManager` will ensure that the JMS broker commits or rolls back.

It's now time to see how this works in practice. In the next section, you'll fix the lost-message problem by adding transactions.

12.2.2 Adding transactions

At the end of section 12.1, you left Rider Auto Parts with the problem of losing messages because you didn't use transactions. Your task now is to apply transactions, which should remedy the problem.

You'll start by introducing Spring transactions to the XML file and adjusting the configuration accordingly. The following listing shows how this is done.

Listing 12.4 XML configuration using Spring transactions

```
<bean id="activemq"
      class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="transacted" value="true"/>    ❶
```

❶ Enables transacted-acknowledge mode

```
    <property name="transactionManager" ref="txManager"/>
  <property name="cacheLevelName" value="CACHE_CONSUMER"/>    ❷
```

❷ Enables consumer cache level

```
</bean>

<bean id="txManager"    ❸
```

❸ Configures Spring JmsTransactionManager

```
      class="org.springframework.jms.connection.JmsTransactionManager">
    <property name="connectionFactory" ref="jmsConnectionFactory"/>
</bean>

<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>    ❹
```

❹

URL to the ActiveMQ message broker

```
  </bean>
```

The first thing you do is turn on `transacted` for the ActiveMQ component ❶, which instructs it to use transacted-acknowledge mode. Then you need to refer to the transaction manager, Spring `JmsTransactionManager` ❸, which manages transactions when using JMS messaging.

JMS, JDBC, AND JTA TRANSACTIONMANAGERS

`JmsTransactionManager` is provided out of the box from the Spring JMS component and is able to handle transactions only with JMS resources. Section 12.3.3 covers using JDBC as a transaction instead of using JMS. Section 12.3.2 covers using both JMS and JDBC together, which requires using the JTA transaction manager.

The JMS transaction manager needs to know how to connect to the JMS broker, which refers to the connection factory. In the `jmsConnectionFactory` definition, you configure the `brokerURL` ❹ to point at the JMS broker.

Because you enabled the transaction ❶, the default behavior of the JMS component is to let the JMS consumer be in autocaching mode. The caching mode has a high impact on the performance, and it's highly recommended to configure the mode to be a cache consumer ❷. This can safely be done with most JMS message brokers such as ActiveMQ. Notice that when using JTA transactions the consumer can't always be cached. The level of cache that can be in use depends on the involved message brokers, databases, and transaction managers. A rule of thumb is to always cache the consumer for non-JTA transactions, and use auto mode for other combinations.

TIP The `cacheLevelName` option on the ActiveMQ component significantly affects performance. We recommend that you read what we just covered one more time.

So far, you've reconfigured only beans in the XML file, which is mandatory when using Spring. In Camel itself, you haven't yet configured anything in relation to transactions. Camel offers great convention over configuration for transaction support, so all you have to do is add `<transacted/>` to the route, after `<from>`, as highlighted here:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <propertyPlaceholder id="properties"
                       location="camelinaction/sql.properties"/>
  <route id="partnerToDB">
    <from uri="activemq:queue:partners"/>
    <transacted/>
    <bean ref="partner" method="toMap"/>
    <to uri="sql:{{sql-insert}}?dataSource=#myDataSource"/>
  </route>
</camelContext>
```

When you specify `<transacted/>` in a route, Camel uses transactions for that particular route and any other routes that the message may undertake. Here in the beginning you'll use transactions with only one route; in section 12.3.5, you'll go further down the road and cover transactions with multiple routes.

When a route is specified as `<transacted/>`, then under the hood Camel looks up the Spring transaction manager and uses it. This is the convention over configuration kicking in.

Using `transacted` in the Java DSL is just as easy, as shown here:

```
from("activemq:queue:partners").routeId("partnerToDB")
  .transacted()
  .bean("partner", "toMap")
  .to("sql:{{sql-insert}}?dataSource=#myDataSource");
```

The convention over configuration applies only when you have a single Spring transaction manager configured. In more complex scenarios, where you either use multiple transaction managers or transaction propagation policies, you have to do additional configuration.

LOOK AT THE SOURCE CODE TO BETTER UNDERSTAND HOW ALL THIS FITS TOGETHER

The book's source code contains this example in the chapter12/riderautoparts-partner directory. At this time, you may want to open the source code in your Java editor and take a look to help you better understand how the configuration in the Spring XML file fits together. In the following section, you'll learn how to test this example.

In this example, all you had to do to configure Camel was to add `<transacted/>` in the route. You relied on the transactional default configurations, which greatly reduces the effort required to set up the various bits. Section 12.3 delves deeper into configuring transactions.

Let's see if this configuration is correct by testing it.

12.2.3 Testing transactions

When you test Camel routes using transactions, it's common to test with live resources, such as a real JMS broker and a database. For example, the book's source code uses Apache ActiveMQ and Derby as live resources. We picked these because they can be easily downloaded using Apache Maven and they're lightweight and embeddable, which makes them perfect for unit testing. No up-front work is needed to install them. To demonstrate how this works, we'll return to the Rider Auto Parts example.

The last time you ran a unit test, you lost the message when there was no connection to the database. Let's try that unit test again, but this time with transactional support. You can do this by running the following Maven goal from the chapter12/riderautoparts-partner directory:

```
mvn test -Dtest=RiderAutoPartsPartnerTransactedTest
```

When you run the unit test, you'll notice a lot of stacktraces printed on the console, and they'll contain the following snippet:

```
2017-10-22 10:08:38,168 [sumer[partners]] WARN  TransactionErrorHandler
java.net.ConnectException: Cannot connect to the database
2017-10-22 10:08:38,173 [sumer[partners]] WARN  EndpointMessageListener
```

You can tell from the stacktrace that `TransactionErrorHandler` (shown in bold) logged an exception at the `WARN` level, with a transaction roll-back message. Just below the `EndpointMessageListenener` (also shown in bold) is logged a `WARN` message with the caused exception and stack-trace. The `EndpointMessageListener` is a `javax.jms.MessageListener`, which is invoked when a new message arrives on the JMS destination. It'll roll back the transaction if an exception is thrown.

Where is the message now? It should be on the JMS queue, so let's add a little code to the unit test to check that.

Uncomment the following code at the end of the unit test method with the name `testNoConnectionToDatabase` (in the `RiderAutoPartsPartnerTransactedTest` class) in listing 12.3:

```
Object body = consumer.receiveBodyNoWait("activemq:queue:partners");
assertNotNull("Should not lose message", body);
```

Now you can run the unit test to ensure that the message wasn't lost—and the unit test will fail with this assertion error:

```
java.lang.AssertionError: Should not lose message
    at org.junit.Assert.fail(Assert.java:74)
    at org.junit.Assert.assertTrue(Assert.java:37)
    at org.junit.Assert.assertNotNull(Assert.java:356)
    at
camelinaction.RiderAutoPartsPartnerTransactedTest.testNoConnectionTo
Database(RiderAutoPartsPartnerTransactedTest.java:97)
```

You're using transactions, and they've been configured correctly, but the message is still being lost. What's wrong? If you dig into the stacktraces,

you'll discover that the message is always redelivered six times, and then no further redelivery is conducted.

TIP If you're using Apache ActiveMQ, we recommend you pick up a copy of *ActiveMQ in Action*, by Bruce Snyder et al. (Manning, 2011).

What happens is that ActiveMQ performs the redelivery according to its default settings, which say it will redeliver at most six times before giving up and moving the message to a dead letter queue. This is, in fact, the Dead Letter Channel EIP. You may remember that we covered this in chapter 11 (look back to figure 11.4). ActiveMQ implements this pattern, which ensures that the broker won't be doomed by a poison message that can't be successfully processed and that would cause arriving messages to stack up on the queue.

NOTE Section 12.3.4 covers transaction redeliveries and nuances in much more depth (for example, when using ActiveMQ as the message broker).

Instead of looking for the message on the partner's queue, you should look for the message in the default ActiveMQ dead letter queue, which is named `ActiveMQ.DLQ`. If you change the code accordingly (as shown in bold), the test will pass:

```
Object body = consumer.receiveBody("activemq:queue:ActiveMQ.DLQ", 5000);
assertNotNull("Should be in ActiveMQ DLQ", body);
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

You need to do one additional test to cover the situation where the connection to the database fails only at first, but works on subsequent calls. Here's that test.

Listing 12.5 Testing a simulated rollback on the first try and a commit on the second try

```
public void testFailFirstTime() throws Exception {
    RouteBuilder rb = new RouteBuilder() {
```

```
      public void configure() throws Exception {
        interceptSendToEndpoint("sql:*")
    .choice()    ❶
```

❶

Causes failure first time

```
            .when(header("JMSRedelivered").isEqualTo("false"))
              .throwException(new ConnectException(
                              "Cannot connect to the database"));
        }
    };

    context.getRouteDefinition("partnerToDB")
        .adviceWith(context, rb);    ❷
```

❷

Advises the route with the simulated error route from ❶

```
    NotifyBuilder notify = new NotifyBuilder(context)
                    .whenDone(1 + 1).create();    ❸
```

❸

Sets up NotifyBuilder for the number of messages you expect

```
    String sql = "select count(*) from partner_metric";
    assertEquals(0, jdbc.queryForInt(sql));

    String xml = "<?xml version=\"1.0\"?>
                + <partner id=\"123\"><date>201702250815</date>
                + <code>200</code><time>4387</time></partner>";

    template.sendBody("activemq:queue:partners", xml);
```

```
    assertTrue(notify.matches(10, TimeUnit.SECONDS));  ❹
```

❹

Waits for routing to be done

```
        assertEquals(1, jdbc.queryForInt(sql));

        Object dlq = consumer.receiveBody("activemq:queue:ActiveMQ.DLQ", 100
    assertNull("Should not be in the DLQ", dlq);  ❺
```

❺

Asserts message not in DLQ

```
    }
```

The idea is to throw `ConnectionException` only the first time. You do this by relying on the fact that any message consumed from a JMS destination has a set of standard JMS headers, and the `JMSRedelivered` header is a `boolean` type indicating whether the JMS message is being redelivered.

The interceptor logic is done in a Camel `RouteBuilder`, so you have the full DSL at your disposal. You use the Content-Based Router EIP ❶ to test the `JMSRedelivered` header and throw the exception only if it's `false`, which means it's the first delivery. The route must then be advised ❷ to use the route that simulates the error. The rest of the unit test should verify correct behavior, so you first check that the database is empty before sending the message to the JMS queue. Then with help from `NotifyBuilder`, you wait for the route to be done ❹, expecting two messages in total ❸; the first message, which should fail, and then the second redelivered message that should complete. After completion, you check that the database has one row. Because you previously were tricked by the JMS broker's dead letter queue, you also check that it's empty ❺.

EXAMPLE FOR APACHE KARAF USERS

The RiderAutoParts Partner example that was used in this section is also provided as an example for users of Apache Karaf, ServiceMix, or JBoss Fuse. The book's source code contains the example in the rider-autoparts-partner-karaf directory. The source includes a readme.md file, which details how to install and try this example on Apache Karaf.

The preceding example uses *local transactions*, because they're based on using only a single resource in the transaction; Spring was orchestrating only the JMS broker. But there was also the database resource, which in the example wasn't under transactional control. Using both the JMS broker and the database as resources participating in the same transaction requires more work, and the next section explains about using single and multiple resources in a transaction. First, we'll look at this from the EIP perspective.

12.3 The Transactional Client EIP

The Transactional Client EIP distills the problem of how a client can control transactions when working with messaging. It's depicted in figure 12.5.

Figure 12.5 A transactional client handles the client's session with the receivers so the client can specify transaction boundaries that encompass the receiver.

Figure 12.5 shows how this pattern was portrayed in Gregor Hohpe and Bobby Woolf's *Enterprise Integration Patterns* book, so it may be a bit difficult to understand how it relates to using transactions with Camel. What the figure shows is that both a sender and a receiver can be transactional by working together. When a receiver initiates the transaction, the message is neither sent nor removed from the queue until the transaction is committed. When a sender initiates the transaction, the message isn't available to the consumer until the transaction has been committed. Figure 12.6 illustrates this principle.

Figure 12.6 A message is being moved from queue A to queue B. Transactions ensure that the message is moved in what appears to be an atomic operation.

The top section of figure 12.6 illustrates the route using EIP icons, with a message being moved from queue A to B using a transaction. The remainder of the figure shows a use-case when one message is being moved.

The middle section shows a snapshot in time when the message is being moved. The message still resides in queue A and hasn't yet arrived in queue B. The message stays in queue A until a commit is issued, which ensures that the message isn't lost in case of a severe failure.

The bottom section shows the situation when a commit has been issued. The message is then deleted from queue A and inserted into queue B. Transactional clients make this whole process appear as an atomic, isolated, and consistent operation.

When talking about transactions, you need to distinguish between single- and multiple-resource transactions. The former are also known as *local* transactions, and the latter as *global* transactions. In the next two sections, we'll look at these two flavors.

12.3.1 Using local transactions

Figure 12.7 depicts the situation of using a single resource—the JMS broker. In this situation, `JmsTransactionManager` orchestrates the transaction with the single participating resource, which is the JMS broker ❶. `JmsTransactionManager` from Spring can orchestrate only JMS-based resources, so the database isn't orchestrated.

In the Rider Auto Parts example in section 12.1, the database didn't participate as a resource in the transaction, but the approach seemed to work anyway. That's because if the database decides to roll back the transaction, it will throw an exception that the Camel `TransactionErrorHandler` propagates back to `JmsTransactionManager`, which reacts accordingly and issues a rollback ❷.

Figure 12.7 Using `JmsTransactionManager` as a single resource in a transaction. The database isn't a participant in the transaction.

This scenario isn't exactly equivalent to enrolling the database in the transaction, because it still has failure scenarios that could leave the system in an inconsistent state. For example, the JMS broker could fail after the database is successfully updated, but before the JMS message is committed. To be absolutely sure that both the JMS broker and the database are in sync in terms of the transaction, you must use global transactions. Let's take a look at that now.

12.3.2 Using global transactions

Using transactions with a single resource is appropriate when a single resource is involved. But the situation changes when you need to span multiple resources in the same transaction, such as JMS and JDBC resources, as depicted in figure 12.8.

In this figure, we've switched to using `JtaTransactionManager`, which handles multiple resources. Camel consumes a message from the queue, and a `begin` is issued ❶. The message is processed, updating the database, and it completes successfully ❷.

What is `JtaTransactionManager`, and how is it different from `JmsTransactionManager` used in the previous section (figure 12.7)? To answer this, you first need to learn a bit about global transactions and where the Java Transaction API (JTA) fits in.

In Java, JTA is an implementation of the XA standard protocol, which is a global transaction protocol. To be able to use XA, the resource drivers must be XA-compliant, which some JDBC and most JMS drivers are. JTA is part of the Java EE specification, which means that any Java EE–compliant application server must provide JTA support. This is one of the benefits of Java EE servers, which have JTA out of the box, unlike some lightweight alternatives, such as Apache Tomcat.

Figure 12.8 Using `JtaTransactionManager` with multiple resources in a transaction. Both the JMS broker and the database participate in the transaction.

JTA is also available in OSGi containers such as Apache Karaf, ServiceMix, or JBoss Fuse. Using JTA outside a Java EE server takes some work to set up because you have to find and use a JTA transaction manager, such as one of these:

- *Atomikos (external third party)*—http://www.atomikos.com
- *Narayana (JBoss AS/WildFly)*—http://narayana.io
- *Apache Geronimo (Java EE)*—http://geronimo.apache.org
- *Apache Aries (OSGi platform)*—http://aries.apache.org

Then you need to figure out how to install and use it in your container and unit tests. The good news is that using JTA with Camel and Spring is just a matter of configuration.

NOTE For more information on JTA, see the Wikipedia page on the subject: http://en.wikipedia.org/wiki/Java_Transaction_API. XA is also briefly discussed here: http://en.wikipedia.org/wiki/X/Open_XA.

When using JTA (XA), there are a couple of differences from using local transactions. First, you have to use XA-capable drivers, which means you have to use `ActiveMQXAConnectionFactory` to let ActiveMQ participate in global transactions:

```
<bean id="jmsXaConnectionFactory"
       class="org.apache.activemq.ActiveMQXAConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

The same applies for the JDBC driver—you need to use an XA-capable driver. You can use Atomikos to set up a pooled XA `DataSource` using an in-memory embedded Apache Derby database:

```
  <bean id="myDataSource" class="com.atomikos.jdbc.AtomikosDataSourceBea
        init-method="init" destroy-method="close">
    <property name="uniqueResourceName" value="partner"/>
    <property name="xaDataSourceClassName"
```

```
                  value="org.apache.derby.jdbc.EmbeddedXADataSource"/>
      <property name="minPoolSize" value="1"/>
      <property name="maxPoolSize" value="5"/>
      <property name="xaProperties">
        <props>
          <prop key="databaseName">memory:partner;create=true</prop>
        </props>
      </property>
    </bean>
```

In a real production system, you should use the JDBC driver of the vendor of your database, such as Oracle, Postgres, MySQL, or Microsoft SQL Server, that's XA-capable.

Having configured the XA drivers, you also need to use the Spring `JtaTransactionManager`. It should refer to the real XA transaction manager, which is Atomikos in this example:

```
  <bean id="jtaTransactionManager"
        class="org.springframework.transaction.jta.JtaTransactionManager"
      <property name="transactionManager" ref="atomikosTransactionManager"
      <property name="userTransaction" ref="atomikosUserTransaction"/>
  </bean>
```

The remainder of the configuration involves configuring Atomikos itself, which you can see in the book's source code, in the file chapter12/xa/src/test/resources/camel-spring.xml.

Suppose you want to add a step to the route shown in figure 12.8. You'll process the message *after* it's been inserted into the database. This additional step will influence the outcome of the transaction, whether or not it throws an exception.

Suppose it does indeed throw an exception, as portrayed in figure 12.9. In this figure, the message is being routed ❶ and, at the last step in the route (in the bottom-right corner with the X), it fails by throwing an exception. `JtaTransactionManager` handles this by issuing rollbacks ❷ to both the JMS broker and the database. Because this scenario uses global transactions, both the database and the JMS broker will roll back, and the final result is as if the entire transaction hadn't taken place.

Figure 12.9 A failure to process a message at the last step in the route causes `JtaTransactionManager` to issue rollbacks to both the JMS broker and the database.

The source code for the book contains this example in the chapter12/xa directory. You can test it using the following Maven goals:

```
mvn test -Dtest=XACommitTest
mvn test -Dtest=XARollbackBeforeDbTest
mvn test -Dtest=XARollbackAfterDbTest
mvn test -Dtest=SpringXACommitTest
mvn test -Dtest=SpringXARollbackBeforeDbTest
mvn test -Dtest=SpringXARollbackAfterDbTest
```

APACHE KARAF EXAMPLE WITH GLOBAL TRANSACTIONS

The example is also available for Apache Karaf, ServiceMix, or JBoss Fuse users in the chapter12/xa-karaf directory, which includes a readme.md file that details how to try the example. Because setting up global transactions with Apache Karaf is difficult, it's highly recommended to study this source code and pay attention to how transaction managers, JCA resources, and the like are defined and configured.

So far, all our examples start from a JMS resource. What if a database is the starting resource?

12.3.3 Transaction starting from a database resource

Transactions can also begin from a database, and that's the topic for this section.

Now suppose you flip the JMS broker and database from the use-case in figure 12.8, which gives you figure 12.10.

Figure 12.10 Using `JtaTransactionManager` with multiple resources in a transaction. The transaction starts from a database resource and includes the JMS broker.

Because you're using both a database and a JMS resource in the same transaction, you need to use `JtaTransactionManager` to orchestrate the transaction. The setup of the database and JMS resources in the Camel ap-

plication is exactly the same as the examples from the previous section. What changes is the Camel route to start from a database.

The book's source code contains this example in the chapter12/xa-database directory. You can try this example by using the following goals:

```
mvn test -Dtest=SpringXACommitTest
mvn test -Dtest=SpringXARollbackBeforeActiveMQTest
mvn test -Dtest=SpringXARollbackAfterActiveMQTest
```

The example uses the SQL component to poll the database for new data:

```
<route>
    <from uri="sql:{{sql-from}}?consumer.onConsume={{sql-delete}}
              &amp;dataSource=#myDataSource&amp;transacted=true"/>
    <transacted/>
    <log message="*** transacted ***"/>
    <to uri="log:row"/>
    <to uri="activemq:queue:order"/>
</route>
```

Notice that the SQL endpoint is configured with `transacted=true`, which tells Camel that the consumer runs in transacted mode. This must be done because the SQL consumer is now the input of the route, and the consumer must be aware of the transaction being used as well. This is similar to when using JMS, where you configured transacted-acknowledge mode, as shown in listing 12.4.

Only a database

If you take out the JMS broker from the example so you're using only the database as a single resource, you don't need to use XA. And instead of using `JtaTransactionManager`, you can use `DataSourceTransactionManager`, which is intended for a single-resource database. The book's source code contains such an example in the chapter12/tx-database directory, and you can try the example by using the following goals:

```
mvn test -Dtest=SpringCommitTest
mvn test -Dtest=SpringRollbackTest
```

NOTE We recommend that WildFly users using the resources that ship with the WildFly server look at the wildfly-camel documentation explaining how to use transactions with WildFly and Apache Camel: http://wildfly-extras.github.io/wildfly-camel/#_jms.

When a message is rolled back, the ACID principle of the transaction means that the outcome is as if the message didn't happen; it's all or nothing. What happens next is covered in the next section.

12.3.4 Transaction redeliveries

When a transaction rolls back, what happens next depends on whether the message came from a message broker or a database. The former will often have various redelivery settings you can configure, which we cover in this section. But neither a database nor the JDBC components from Camel provide transactional redelivery support. At the end of this section, we'll talk about what you can do instead.

On a message broker such as Apache ActiveMQ, message redelivery can be handled and configured at two levels:

- Consumer level (used by default)
- Broker level

By default, redelivery is controlled by the consumer: it's the duty of the consumer to keep track of the number of times a message has been redelivered, the amount of time the consumer should delay between redelivery attempts, and whether a message is exhausted and should be moved to the ActiveMQ dead letter queue.

ACTIVEMQ REDELIVERY VS. CAMEL REDELIVERY

This may feel familiar, and, yes, it's also similar to what Camel's error handler is capable of doing. Many of the options you use to configure redelivery with ActiveMQ are similar to options you'd use with Camel. This isn't a surprise, as both ActiveMQ and Camel were created originally by the same group of people.

To allow the consumer to keep track of redelivery of the message, the consumer must be reused by the Camel route, and to ensure that you must configure the JMS or ActiveMQ component in Camel to cache the consumer, as shown here in bold:

```
<bean id="activemq"
      class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="transacted" value="true"/>
    <property name="transactionManager" ref="jtaTransactionManager"/>
    <property name="cacheLevelName" value="CACHE_CONSUMER"/>
</bean>
```

This is important to remember, because the JMS or ActiveMQ component will by default be conservative in transacted mode, and not cache the consumer. You may not notice a difference because when a transaction rollback happens, the message is still being redelivered. But because the consumer wasn't cached, a new consumer is created for each message received, and that consumer has no previous state about the redelivery, and as a result there are no delays between any redeliveries. Because there are no delays between redeliveries, the redelivered messages will happen rapidly in sequence. The ActiveMQ also keeps track of the number of redelivery attempts, and when the message has failed too many times in a row, it's automatically moved to the ActiveMQ dead letter queue.

Configuring consumer-level redelivery with ActiveMQ

But if the consumer is cached, then by default the consumer will delay each redelivery by 1 second and therefore redeliveries will be a bit slower. Because 1 second may still be too fast, you can configure a slower setting in the `brokerURL` option, as shown here:

```
<bean id="jmsXaConnectionFactory"
      class="org.apache.activemq.ActiveMQXAConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616
              &amp;jms.redeliveryPolicy.maximumRedeliveries=10
              &amp;jms.redeliveryPolicy.redeliveryDelay=5000"/>
</bean>
```

Each redelivery option must be prefixed with `jms.redeliveryPolicy`. In the preceding example, we configured maximum redeliveries to 10, and

used 5 seconds as delay.

TIP ActiveMQ has other ways to configure redelivery, which you can find more about on the website: http://activemq.apache.org/redelivery-policy.html.

Consumer-level redelivery conducts retries so that message ordering is maintained while messages appear as inflight on the broker. Redelivery of messages from the given queue is limited to a single consumer, as the broker is unaware that the consumer is performing redeliveries. If message ordering isn't important, and/or higher throughput and load distribution among concurrent consumers is desired, then consumer level redelivery isn't sufficient, and broker level should be used instead.

Configuring broker-level redelivery with ActiveMQ

Broker-level redelivery is configured as an ActiveMQ plugin that's defined in the broker XML configuration file:

```
<broker xmlns="http://activemq.apache.org/schema/core"
        schedulerSupport="true">
  <plugins>
    <redeliveryPlugin fallbackToDeadLetter="true"
                      sendToDlqIfMaxRetriesExceeded="true">
      <redeliveryPolicyMap>
        <redeliveryPolicyEntries>
          <defaultEntry>
            <redeliveryPolicy maximumRedeliveries="10"
                              initialRedeliveryDelay="5000"
                              redeliveryDelay="10000"/>
          </defaultEntry>
      </redeliveryPolicyMap>
    <redeliveryPlugin>
  </plugins>
```

In this example, we perform at most 10 redeliveries with 10 seconds delay in between, but the first delay is only 5 seconds. You can find more details about ActiveMQ redelivery at the ActiveMQ website.

Let's now talk about redeliveries when you start from a database.

Transaction redeliveries starting from database

As opposed to a message broker, a database doesn't have the concept of redelivery or dead letter queue. Therefore, you have limited options of what you can do to address repeated transaction rollbacks when starting from a database.

The book's source code contains an example that starts a transaction from a database in the chapter12/tx-database directory. There's a rollback example that you can try using the following Maven goal:

```
mvn test -Dtest=SpringRollbackTest
```

The SQL consumer will, by default, poll the database every half second. Because the example is designed to fail, it'll throw an exception during processing that causes a transactional rollback. On the next poll, the same thing happens again, and again, and so on. The repeated number of consecutive rollbacks may stress the database and your monitoring system, which may detect a failure every half second. In those situations, you can configure the SQL consumer to back off polling so frequently, as shown here:

```
<from uri="sql:{{sql-from}}?consumer.onConsume={{sql-delete}}
          &amp;dataSource=#myDataSource&amp;transacted=true
          &amp;backoffMultiplier=5&amp;backoffErrorThreshold=1"/>
```

The options `backoffErrorThreshold=1` and `backoffMultiplier=5` tell Camel to multiply the polling interval by 5 after there's been one error, so the polling will happen every 0.5 × 5 = 2.5 seconds. You want to use this when the error may be recoverable, and maybe after a while the message can be processed successfully, and the transaction can commit. When this happens, the back-off will return to normal, and poll every half second again.

TIP The SQL endpoint also provides a back-off capability when there's no data, using the `backoffIdleThreshold` option. For example, `backoffIdleThreshold=3` will trigger back-off after three consecutive polls with no data.

So far, we've used convention over configuration when configuring transactions in Camel, by just adding `<transacted/>` to the route. This is often all you'll need to use, but at times you'll need more fine-grained control, such as using two different transaction propagations.

12.3.5 Using different transaction propagations

In some situations, you may need to use multiple transactions with the same exchange, as illustrated in figure 12.11.

Figure 12.11 Using two independent transactions in a single exchange

In figure 12.11, an exchange is being routed in Camel. It starts off using the required transaction, and then you need to use another transaction that's independent of the existing transaction. You can do this by using `PROPAGATION_REQUIRES_NEW`, which will start a new transaction regardless of whether an existing transaction exists. When the exchange completes, the transaction manager will issue commits or rollbacks to these two transactions, which ensures that they both complete at the same time. Because two transaction legs are in play, they can have different outcomes; for example, transaction 1 can roll back, while transaction 2 commits, and vice versa.

In Camel, a route can be configured to use only at most one transaction propagation, which means figure 12.11 must use two routes. The first route uses `PROPAGATION_REQUIRED`, and the second route uses `PROPAGATION_REQUIRES_NEW`.

Suppose you have an application that updates orders in a database. The application must store all incoming orders in an audit log and then it either updates or inserts the order in the order database. The audit log should always insert a record, even if subsequent processing of the order fails. Implementing this in Camel should be done using two routes, as shown in the following listing.

Listing 12.6 Using two independent transactions in a single exchange

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

```
      <from uri="activemq:queue:inbox"/>
    <transacted ref="required"/>  (1)
```

(1) Requires transaction

```
    <to uri="direct:audit"/>  (2)
```

(2) Calls the audit-log route

```
    <to uri="direct:order"/>  (3)
```

(3) Calls the insert-order route

```
      <to uri="activemq:queue:order"/>
    </route>

    <route>
      <from uri="direct:audit"/>
    <transacted ref="requiresNew"/>  (4)
```

(4) Uses a new transaction for the audit-log route

```
      <bean ref="auditLogService" method="insertAuditLog"/>
    </route>

    <route>
      <from uri="direct:order"/>
    <transacted ref="mandatory"/>  (5)
```

❺

Uses the existing parent transaction for the insert-order route

```
      <bean ref="orderService" method="insertOrder"/>
    </route>
  </camelContext>
```

The first route uses `PROPAGATION_REQUIRED` to start transaction 1 ❶. Then the audit-log route is called ❷, which uses `PROPAGATION_REQUIRES_NEW` ❹ to start transaction 2 that runs independently from transaction 1. After the message has been inserted into the audit log, the insert-order route is called ❸. This route will reuse the existing transaction 1 from the parent route ❺, which is specified by using the mandatory propagation property.

In listing 12.6, we're using three propagation behaviors, which are configured as shown in the following listing.

Listing 12.7 Configure three propagation behavior beans in XML DSL

```
<bean id="required"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
  <property name="transactionManager" ref="jtaTransactionManager"/>
  <property name="propagationBehaviorName" value="PROPAGATION_REQUIRED"/
</bean>

<bean id="requiresNew"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
  <property name="transactionManager" ref="jtaTransactionManager"/>
  <property name="propagationBehaviorName"
            value="PROPAGATION_REQUIRES_NEW"/>
</bean>

<bean id="mandatory"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
  <property name="transactionManager" ref="jtaTransactionManager"/>
  <property name="propagationBehaviorName" value="PROPAGATION_MANDATORY"
</bean>
```

The book's source code contains this example in the chapter12/propagation directory, which you can try using the following

Maven goal:

```
mvn test -Dtest=PropagationTest
mvn test -Dtest=PropagationRollbackLastTest
mvn test -Dtest=SpringPropagationTest
mvn test -Dtest=SpringPropagationRollbackLastTest
```

Four unit tests demonstrate different scenarios, as described in table 12.1.

Table 12.1 Overview of what happens when transaction 1 or 2 either commits or rolls back

Test method	Transaction 1	Transaction 2	Comment
`testWithCamel`	Commit	Commit	The message is successfully processed. In the database, one row is inserted in both the order and audit-log tables.
`testWithDonkey`	Rollback	Commit	The message fails during order validation and is rolled back. In the database, no row is inserted in the order table. But six rows are inserted in the audit-log table because transaction 2 commits. This is intended, as the use-case was designed to

Test method	Transaction 1	Transaction 2	Comment
			insert all orders into audit logs even if transaction 1 fails.
`testAuditLogFail`	Rollback	Rollback	The message fails during audit logging and is rolled back. In the database, no row is inserted, as both transactions roll back.
`testAuditLog-RollbackLast`	Commit	Rollback	The message fails during audit logging, and only the second transaction is explicitly marked to roll back, leaving the first transaction to commit. Only the order is inserted into

Test method	Transaction 1	Transaction 2	Comment
			the database, whereas the audit log fails and is rolled back.

KEEP IT SIMPLE

Although you can use multiple propagation behaviors with multiple routes in Camel, do so with care. Try to design your solutions with as few propagations as possible, because complexity increases dramatically when you introduce new propagation behaviors. For example, table 12.1 lists four outcomes of using two propagations; if you have three propagations, there are $2^3 = 8$ combinations.

In the next section, we'll return to Rider Auto Parts and look at an example that covers a common use-case: using web services together with transactions. How do you return a custom web service response if a transaction fails?

12.3.6 Returning a custom response when a transaction fails

Rider Auto Parts has a Camel application that exposes a web service to a selected number of business partners. The partners use this application to submit orders. Figure 12.12 illustrates the application.

Figure 12.12 A web service used by business partners to submit orders. A copy of the order is stored in a database before it's processed by the ERP system.

As you can see, the business partners invoke a web service to submit an order ❶. The received order is stored in a database for audit purposes ❷. The order is then processed by the enterprise resource planning (ERP) system ❸, and a reply is returned to the waiting business partner ❹.

The web service is deliberately kept simple so partners can easily use it with their IT systems. A single return code indicates whether the order

succeeded or failed. The following code snippet is part of the WSDL definition for the reply (`outputOrder`):

```
<xs:element name="outputOrder">
    <xs:complexType>
        <xs:sequence>
            <xs:element type="xs:string" name="code"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

The `code` field should contain `OK` if the order was accepted; any other value is considered a failure. The Camel application must deal with any thrown exceptions and return a custom failure message, instead of propagating the thrown exception back to the web service.

Your Camel application needs to do the following three things:

- Catch the exception and handle it to prevent it from propagating back
- Mark the transaction to roll back
- Construct a reply message with a `code` value of `ERROR`

Camel can support such complex use-cases because you can use `onException`, which you learned about in chapter 11.

Catching and handling the exception

What you do is add an `onException` to the `Camel-Context`, as shown here:

```
<onException>
    <exception>java.lang.Exception</exception>
    <handled><constant>true</constant></handled>
    <transform><method bean="order" method="replyError"/></transform>
    <rollback markRollbackOnly="true"/>
</onException>
```

You first tell Camel that this `onException` should trigger for any kind of exception that's thrown. You then mark the exception as `handled`, which removes the exception from the exchange, because you want to use a custom reply message instead of the thrown exception.

Rolling back a transaction

To roll back the transaction, you mark it for rollback using a single line of code:

```
<rollback markRollbackOnly="true"/>
```

By doing this, you trigger a rollback without an exception being thrown using the `markRollbackOnly` attribute. The `<rollback/>` definition must always be at the end of `onException` because it stops the message from being further routed. That means you *must* have prepared the reply message before you issue the `<rollback/>`.

ROLLBACK STRATEGIES

You can roll back a transaction in several ways. By default, if any unhandled exception happens during routing, the transaction error handler detects it and stops routing, and the exception is propagated back to the transaction manager that marks the transaction for rollback. Unhandled exceptions trigger a transactional rollback. In the Camel routes, you can make rollbacks obvious by using `rollback` in the DSL. The `rollback` is merely a facade for throwing an exception of the type `org.apache.camel.RollbackExchangeException`. To do this in XML DSL, you can specify the rollback with an optional message:

```
<rollback message="Forced being rolled back"/>
```

And in Java DSL, the same statement is as follows:

```
rollback("Forced being rolled back")
```

Constructing the reply message

To construct the reply message, you use the `order` bean, invoking its `replyError` method:

```
public OutputOrder replyError(Exception cause) {
    OutputOrder error = new OutputOrder();
    error.setCode("ERROR: " + cause.getMessage());
```

```
        return error;
    }
```

This is easy to do, as you can see. You first define the `replyError` method to have an `Exception` as a parameter—this will contain the thrown exception. You then create the `OutputOrder` object, which you populate with the `ERROR` text and the exception message.

The book's source code contains this example in the chapter12/riderautoparts-order directory. You can start the application using the following Maven goal:

```
mvn camel:run
```

Then you can send web service requests to http://localhost:9000/order. The WSDL is accessible at http://localhost:9000/order?wsdl.

To work with this example, you need to use web services. SoapUI (www.soapui.org) is a popular application for testing with web services. It's also easy to set up and get started. You create a new project and import the WSDL file from http://localhost:9000/order?wsdl. Then you create a sample web service request and fill in the request parameters, as shown in figure 12.13. You then send the request by clicking the green Play button, and it will display the reply in the pane on the right side.

Figure 12.13 A web service message causes the transaction to roll back, and a custom reply message is returned.

In the example in figure 12.13, we caused a failure to occur. Our example behaves according to what you specify in the `refNo` field. You can force different behavior by specifying either `FATAL` or `FAIL-ONCE` in the `refNo` field. Entering any other value will cause the request to succeed. As figure 12.13 shows, we entered `FATAL`, which causes an exception to occur and an `ERROR` reply to be returned.

So far, we've been using resources that support transactions, such as JMS and JDBC, but the majority of components don't support transactions. What can you do instead? The next section looks at compensating when transactions aren't supported.

12.4 Compensating for unsupported transactions

The number of resources that can participate in transactions is limited; they're mostly confined to JMS- and JDBC-based resources. This section covers what you can do to compensate for the absence of transactional support in other resources. *Compensation*, in Camel, involves the unit-of-work concept.

First, you'll look at how a unit of work is represented in Camel and how you can use this concept. Then we'll walk through an example demonstrating how the unit of work can help simulate the orchestration of a transaction manager. We'll also discuss how you can use a unit of work to compensate for the lack of transactions by doing the work that a transaction manager's rollback would do in the case of failure.

12.4.1 Introducing UnitOfWork

Conceptually, a *unit of work* is a batch of tasks as a single coherent unit. The idea is to use the unit of work as a way of mimicking transactional boundaries.

In Camel, a unit of work is represented by the `org.apache.camel.spi.UnitOfWork` interface, offering a range of methods, including the following:

```
void addSynchronization(Synchronization synchronization);
void removeSynchronization(Synchronization synchronization);
void done(Exchange exchange);
```

The `addSynchronization` and `removeSynchronization` methods are used to register and unregister a `Synchronization` (a callback), which we'll look at shortly. The `done` method is invoked when the unit of work is complete, and it invokes the registered callbacks.

The `Synchronization` callback is the interesting part for Camel end users because it's the interface you use to execute custom logic when an exchange is complete. It's represented by the `org.apache.camel.spi.Synchronization` interface and offers these two methods:

```
void onComplete(Exchange exchange);
void onFailure(Exchange exchange);
```

When the exchange is done, either the `onComplete` or `onFailure` method is invoked, depending on whether the exchange failed.

Figure 12.14 illustrates how these concepts are related to each other. As you can see, each `Exchange` has exactly one `UnitOfWork`, which you can access using the `getUnitOfWork` method from the `Exchange`. The `UnitOfWork` is private to the `Exchange` and isn't shared with others.

Figure 12.14 An `Exchange` has one `UnitOfWork`, which in turn has from zero to many `Synchronization`s.

HOW UNITOFWORK IS ORCHESTRATED

Camel will automatically inject a new `UnitOfWork` into an `Exchange` when it's routed. This is done by an internal processor, `UnitOfWorkProcessor`, which is involved in the start of every route. When the `Exchange` is done, this processor invokes the registered `Synchronization` callbacks. The `UnitOfWork` boundaries are always at the beginning and end of the `Exchange` being routed.

When an `Exchange` is done being routed, you hit the end boundary of the `UnitOfWork`, and the registered `Synchronization` callbacks are invoked one by one. This is the same mechanism the Camel components use to add their custom `Synchronization` callbacks to the `Exchange`. For example, the File and FTP components use this mechanism to perform *after-processing* operations such as moving or deleting processed files.

TIP Use `Synchronization` callbacks to execute any after-processing you want done when the `Exchange` is complete.

A good way of understanding how this works is to review an example, which we'll do now.

12.4.2 Using synchronization callbacks

Rider Auto Parts has a Camel application that sends email messages containing invoice details to customers. First, the email content is generated, and then, before the email is sent, a backup of the email is stored in a file for reference. Whenever an invoice is to be sent to a customer, the Camel application is involved. Figure 12.15 illustrates the application.

Figure 12.15 Emails are sent to customers listing their invoice details. Before the email is sent, a backup is stored in the filesystem.

Imagine what would happen if there were a problem sending an email. You can't use transactions to roll back, because filesystem resources can't participate in transactions. Instead, you can perform custom logic, which *compensates* for the failure by deleting the file.

The compensation logic is trivial to implement, as shown here:

```
public class FileRollback implements Synchronization {

    public void onComplete(Exchange exchange) {
    }

    public void onFailure(Exchange exchange) {
        String name = exchange.getIn().getHeader(
                              Exchange.FILE_NAME_PRODUCED, String.class)
        LOG.warn("Failure occurred so deleting backup file: " + name);
        FileUtil.deleteFile(new File(name));
    }
}
```

In the `onFailure` method, you delete the backup file, retrieving the filename used by the Camel File component from the `Exchange.FILE_NAME_PRODUCED` header.

TIP Camel offers the `org.apache.camel.support.SynchronizationAdapter` class, which is an empty `Synchronization` implementation having the need for end users to override only the `onComplete` or `onFailure` method as needed.

What you must do next is instruct Camel to use the `FileRollback` class to perform this compensation. To do so, you can add it to the `UnitOfWork` by using the `addSynchronization` method, which was depicted in figure 12.15. This can be done using the Java DSL as highlighted in the following listing.

Listing 12.8 Using `UnitOfWork` as a transactional rollback from inlined `Processor`

```
public void configure() throws Exception {
  from("direct:confirm")
  .process(new Processor() {    ❶
```

❶ Inlined Processor

```
    public void process(Exchange exchange) throws Exception {    ❶
      exchange.getUnitOfWork()    ❶
        .addSynchronization(new FileRollback());    ❶
    }    ❶
  })    ❶
    .bean(OrderService.class, "createMail")
    .log("Saving mail backup file")
    .to("file:target/mail/backup")
    .log("Trying to send mail to ${header.to}")
    .bean(OrderService.class, "sendMail")
    .log("Mail sent to ${header.to}");
}
```

The book's source code contains this example in the chapter12/uow directory. You can try it using the following Maven goal:

```
mvn test -Dtest=FileRollbackTest
```

If you run the example, it will output something like the following to the console:

```
INFO  route1 - Saving mail backup file
INFO  route1 - Trying to send to FATAL
ERROR DefaultErrorHandler - Failed delivery for (MessageId: ID-davsclaus

WARN  FileRollback - Failure occurred so deleting backup file: target/ma
```

One thing that may bother you is that you must use an inlined `Processor` ❶ to add the `FileRollback` class as a `Synchronization`. Camel offers a convenient method on the `Exchange`, so you could do it with less code:

```
exchange.addOnCompletion(new FileRollback());
```

And with Java 8, you can inline the `Processor` ❶ using lambda style, which reduces the code from six lines to only one line:

```
.process(e -> e.addOnCompletion(new FileRollback()))
```

But it still requires the inlined `Processor` ❶ with or without Java 8 lambda style. Isn't there a more convenient way? Yes, and that's where `onCompletion` comes into the picture.

12.4.3 Using onCompletion

`onCompletion` takes the `Synchronization` into the world of routing, enabling you to easily add the `FileRollback` class as `Synchronization`. Let's see how it's done.

`OnCompletion` is available in both Java DSL and XML DSL variations. The following listing shows how `onCompletion` is used with XML DSL.

Listing 12.9 Using `onCompletion` as a transactional rollback

```
<bean id="orderService" class="camelinaction.OrderService"/>
<bean id="fileRollback" class="camelinaction.FileRollback"/>
```

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

  <onCompletion onFailureOnly="true">
```

Context scoped onCompletion that triggers on failure only

```
        <bean ref="fileRollback" method="onFailure"/>
    </onCompletion>

    <route>
        <from uri="direct:confirm"/>
        <bean ref="orderService" method="createMail"/>
        <log message="Saving mail backup file"/>
        <to uri="file:target/mail/backup"/>
        <log message="Trying to send mail to ${header.to}"/>
        <bean ref="orderService" method="sendMail"/>
        <log message="Mail sent to ${header.to}"/>
    </route>
</camelContext>
```

As you can see, `<onCompletion>` is defined as a separate Camel route. `onCompletion` will be executed right after the message is done being routed. In this example, that means after the last step in the route, the message reaches the end, and `onCompletion` is executed:

```
<log message="Mail sent to ${header.to}"/>
```

UNDER THE HOOD

When you add `<onCompletion>` to `<camelContext>` or `<route>`, `<onCompletion>` will be added as a subroute that's being routed from a `Synchronization` callback. The routing engine is responsible for orchestrating this by detecting whether `<onCompletion>` is available, and if so by adding the necessary `Synchronization` callback to the `UnitOfWork` of the exchange. This happens when the exchange is about to be routed to the original route. When the exchange is done being routed, `UnitOfWork` kicks in and calls each `Synchronization` callback, and hence among those the callback that triggers the `<onCompletion>` subroute.

In the present situation, you're interested in executing `onCompletion` only when the exchange fails, so you can specify this by setting the `onFailureOnly` attribute to `true`.

The book's source code contains this example in the chapter12/uow directory; you can run the example using the following Maven goal:

```
mvn test -Dtest=FileRollbackOnCompletionTest
mvn test -Dtest=SpringFileRollbackOnCompletionTest
```

When you run it, you'll find that it acts just like the previous example. It will delete the backup file if the exchange fails.

`onCompletion` can also be used when the exchange didn't fail. Suppose you want to log activity about exchanges being processed. For example, in the Java DSL, you could do it as follows:

```
onCompletion().bean("logService", "logExchange");
```

`onCompletion` also supports scoping, exactly the same as `onException` does at either context or route scope (as you saw in chapter 11). You could create a Java DSL–based version of listing 12.9 using route-scoped `onCompletion` as follows:

```
from("direct:confirm")
    .onCompletion().onFailureOnly()
```

```
            .bean(FileRollback.class, "onFailure")
        .end()
        .bean(OrderService.class, "createMail")
        .log("Saving mail backup file")
        .to("file:target/mail/backup")
        .log("Trying to send mail to ${header.to}")
        .bean(OrderService.class, "sendMail")
        .log("Mail send to ${header.to}");
```

You can also attach a predicate to `onCompletion` to trigger only when the predicate matches.

Using predicate with OnCompletion

Suppose you want `onCompletion` to trigger only if a file was saved. To do this, you can use `onWhen` as the predicate to check for the presence of the `Exchange.FILE_NAME_PRODUCED` header, which the file producer adds as a message header with the name of the file that was saved:

```
from("direct:confirm")
    .onCompletion()
        .onFailureOnly().onWhen(header(Exchange.FILE_NAME_PRODUCED))
        .bean(FileRollback.class, "onFailure")
    .end()
```

And the corresponding code example in XML DSL:

```
<onCompletion onFailureOnly="true">
  <onWhen><header>CamelFileNameProduced</header></onWhen>
  <bean ref="fileRollback" method="onFailure"/>
</onCompletion>
```

Notice you have to use the value of the `Exchange.FILE_NAME_PRODUCED` key (`CamelFileNameProduced`) in XML DSL as the header name.

`onCompletion` runs by default after the consumer is completed. If you want to run `onCompletion` before the consumer returns a response to the caller, you have to switch modes.

Using BeforeConsumer mode

As part of your work for Rider Auto Parts, you recently began developing a simple API for mobile users to access information about their orders. The API exposes a REST service that can return information about an order. Figure 12.16 illustrates this principle.

Figure 12.16 A mobile user accesses a Camel API that fetches order information and enriches the response with a token header.

Figure 12.16 shows how mobile users can access the REST service using the API ❶, which then queries the ❷ order service to obtain order details. If the request and order are valid, the response must be enriched with a token ❸ that the mobile client must use for subsequent queries to the API.

The API is implemented in Camel via a `RouteBuilder` class using rest-dsl, as shown in the following listing.

Listing 12.10 API implemented using rest-dsl in Camel

```
public void configure() throws Exception {

  restConfiguration().component("netty4-http")    ❶
```

❶ Configures rest on localhost:8080/service using JSON binding

```
                      .bindingMode(RestBindingMode.json)
                      .host("localhost").port(8080).contextPath("service"

  onCompletion().modeBeforeConsumer()    ❸
```

❸ OnCompletion that adds the token header to be included in the response

```
      .setHeader("Token").method(TokenService.class);
```

```
    rest()
  .get("/order/{id}")    ❷
```

---

❷

Rest DSL defining a service to get order by ID

```
        .to("bean:orderService?method=getOrder");
```

First, you set up the rest configuration to use the netty4-http component as the HTTP server ❶. You're using Netty because it's a popular, fast, and lightweight networking framework; which works well in these times of microservices. The REST services are exposed on localhost:8080/service and use JSON binding mode.

In Rest DSL, you define a REST service to get the order status by ID ❷, which will map to clients using the URL `http://localhost:8080/service/order/{id}`.

As part of the API, it's now a requirement that a token header is returned with a unique value. The mobile service is then required to use the token header for subsequent access to the API when other services are requested. To add the unique token header, you use `onCompletion` ❸. Switching to `BeforeConsumer` mode ensures that the header is added to the message before the REST service returns the response to the mobile client.

The book's source code contains this example in chapter12/uow as a Java and Spring example. The following Maven goals run the example in either mode:

```
mvn compile exec:java
mvn compile exec:java -Pspring
```

When the example runs, you can use a web browser or REST client like `curl` to call the service. The order ID 123 is hardcoded to return a response, including the token shown in bold:

```
$ curl -i http://localhost:8080/service/order/123
HTTP/1.1 200 OK
Content-Length: 37
Accept: */*
breadcrumbId: ID-davsclaus-pro-63316-1427025423836-0-7
id: 123
Token: 315904563655664740
User-Agent: curl/7.30.0
Content-Type: application/json
Connection: keep-alive

{"id":123,"amount":1,"motor":"Honda"}
```

If you send another request, notice that the token value will change. You can also try to change the example and see what happens if you change the mode to after consumer.

Now you've learned all there is to know about `onCompletion`, which brings us to the next part, where we cover idempotency.

12.5 Idempotency

So far in this chapter, we've talked about how transactions can be used as a means to coordinate state updates among distributed systems to form a unit of work as a whole. Related to this concept is idempotency, the topic for the remainder of this chapter. The term *idempotent* is used in mathematics to describe a function that can be applied multiple times without changing the result beyond the initial result. In computing, *idempotent* is used to describe an operation that will produce the same results if executed once or multiple times.

Idempotency is documented in the EIP book as the Idempotent Consumer pattern. This pattern deals with the problem of how to reliably exchange messages in a distributed system. The book covers the complexity of how a sender can send a message safely to a receiver and many of the situations where something can go wrong. Figure 12.17 shows an example.

Figure 12.17 The receiver receives duplicate messages because of the problem of returning acknowledgements from the receiver to the sender.

In this example, when a sender sends a message to a receiver ❶, the sender can know that the receiver has received and accepted the message only if an acknowledge message is returned from the receiver. But if that acknowledgment is lost due to a networking issue ❷, the sender may need to resend the message that the receiver has already received—a duplicate message ❸.

Therefore, many messaging systems, such as Apache ActiveMQ, have capabilities to eliminate duplicate messages, so the users don't have to worry about duplicates. In Camel, this mechanism is implemented as the Idempotent Consumer EIP.

12.5.1 Idempotent Consumer EIP

In Camel, the Idempotent Consumer EIP is used to ensure routing a message once and only once. To achieve this, Camel needs to be able to detect duplicate messages, which involves the following two procedures:

- Generating a unique key for each message
- Storing and retrieving previously seen keys

The heart of the Idempotent Consumer implementation is the *repository*, which is defined as the interface `org.apache.camel.spi.IdempotentRepository` with the following methods:

```
boolean add(E key);
boolean contains(E key);
boolean remove(E key);
boolean confirm(E key);
```

Figure 12.18 illustrates how this works in Camel, and the following steps detail the process:

1. When an `Exchange` reaches the idempotent consumer, the unique key is evaluated from the `Exchange` by using the configured Camel `Expression`. For example, this can be a message header or an XPath expression.

2. The key is checked against the idempotent repository to see whether it's a duplicate. This is done by calling the `add` method. If the method returns `false`, the key was successfully added and no duplicate was detected. If the method returns `true`, an existing key already exists and the message is detected as a duplicate.
3. If the `Exchange` isn't a duplicate, it's routed as usual.
4. When the `Exchange` is finished being routed, either the commit or rollback method of its `Synchronization` is invoked.
5. To commit the `Exchange`, the `confirm` method is invoked on the idempotent repository. This allows the idempotent repository to do any post cleanup or other necessary tasks when an `Exchange` has been committed.
6. To roll back the `Exchange`, the `remove` method is invoked on the idempotent repository, which removes the key from the repository. Because the key is removed, then upon redelivery of the same message, Camel will have no prior knowledge of having seen this message, and therefore won't regard the message as a duplicate. If you want to prevent this and regard the redelivered message as a duplicate, you can configure the option `removeOnFailure` to `false` on the idempotent repository. Consequently, upon rollback, the key isn't removed but is confirmed (upon rollback, the same action is performed as for a commit—step 5).

Figure 12.18 How the Idempotent Consumer EIP operates in Camel

Let's see how to do this in action.

Using the Idempotent Consumer EIP

In order to use the idempotent consumer EIP, you need to set up the idempotent repository first. The following listing shows an example of how to do this using XML DSL.

Listing 12.11 Using Idempotent Consumer EIP in XML DSL

```
<bean id="repo"     1
```

```
    class="org.apache.camel.processor.idempotent.MemoryIdempotentRepositor

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="seda:inbox"/>
      <log message="Incoming order ${header.orderId}"/>
      <to uri="mock:inbox"/>
    <idempotentConsumer messageIdRepositoryRef="repo">    ❷
```

❶ Using the memory idempotent repository

❷ Route segment using the idempotent consumer

```
      <header>orderId</header>    ❸
```

❸ Expression to evaluate the unique key

```
        <log message="Processing order ${header.orderId}"/>
        <to uri="mock:order"/>
      </idempotentConsumer>
    </route>
  </camelContext>
```

In this example, you use the built-in memory-based repository, which is defined as a bean in XML DSL ❶. In the Camel route, you then wrap the routing segment that you want to be invoked only once per unique message ❷. The attribute `messageIdRepositoryRef` ❷ must be configured to refer to the chosen repository ❶. The example uses the message header `orderId` as the unique key ❸.

NOTE The unique key is evaluated as a `String` type. Therefore, you can use only information from the `Exchange` that can be converted to a `String` type. You can't use complex types such as Java objects unless they generate unique keys using their `toString` method.

The example from listing 12.11 can be written using Java DSL, as in the following listing.

Listing 12.12 Idempotent Consumer using Java DSL

```
public void configure() throws Exception {

  IdempotentRepository repo = new MemoryIdempotentRepository()   ❶
```

❶ Using the in-memory idempotent repository

```
    from("seda:inbox")
      .log("Incoming order ${header.orderId}")
      .to("mock:inbox")
    .idempotentConsumer(header("orderId"), repo)   ❷
```

❷ Route segment using the idempotent consumer, and expression to evaluate unique key

```
        .log("Processing order ${header.orderId}")
        .to("mock:order")
      .end();
  }
```

The book's source code contains this example in the chapter12/idempotent directory. You can try the example using the following Maven goals:

```
mvn test -Dtest=SpringIdempotentTest
mvn test -Dtest=IdempotentTest
```

Running this example will send five messages to the Camel route, two of which will be duplicated, so only three messages will be processed by the idempotent consumer. The example uses the following code to send the five messages:

```
template.sendBodyAndHeader("seda:inbox", "Motor", "orderId", "123");
template.sendBodyAndHeader("seda:inbox", "Motor", "orderId", "123");
template.sendBodyAndHeader("seda:inbox", "Tires", "orderId", "789");
template.sendBodyAndHeader("seda:inbox", "Brake pad", "orderId", "456");
template.sendBodyAndHeader("seda:inbox", "Tires", "orderId", "789");
```

This outputs the following to the console:

```
2017-04-12 [ - seda://inbox] INFO  route1 - Incoming order 123
2017-04-12 [ - seda://inbox] INFO  route1 - Processing order 123
2017-04-12 [ - seda://inbox] INFO  route1 - Incoming order 123
2017-04-12 [ - seda://inbox] INFO  route1 - Incoming order 789
2017-04-12 [ - seda://inbox] INFO  route1 - Processing order 789
2017-04-12 [ - seda://inbox] INFO  route1 - Incoming order 456
2017-04-12 [ - seda://inbox] INFO  route1 - Processing order 456
2017-04-12 [ - seda://inbox] INFO  route1 - Incoming order 789
```

As you can see, the orders are processed only once, indicated by the log `"Processing order ..."`.

The example uses the in-memory idempotent repository, but Camel provides other implementations.

12.5.2 Idempotent repositories

Camel provides many idempotent repositories, each supporting a different level of service. Some are for standalone use in memory only; others support clustered environments. We've listed six of the most commonly used implementations provided by Apache Camel 2.20, with some tips and information about pros and cons next.

MemoryIdempotentRepository

A fast in-memory store that stores entries in a Map structure. All data will be lost when the JVM is stopped. There's no support for clustering. This store is provided in the camel-core module.

FileIdempotentRepository

A file-based store that uses a Map as a cache for fast lookup. All write operations are synchronized, so the store can be a potential bottleneck for high concurrent throughput. Each write operation rewrites the file, which means for large data sets writes can be slower. This store supports active/passive clustering, where the passive cluster is in standby and takes over processing if the master dies. This store is provided in the camel-core module.

JdbcMessageIdRepository

A store using JDBC to store keys in a SQL database. This implementation uses no caching, so each operation accesses the database. Each idempotent consumer must be associated with a unique name but can reuse the same database table to store keys. High-volume processing can become a performance bottleneck if the database can't keep up. Clustering is supported in both active/passive and active/active mode. This store is provided in the camel-sql module. To use this store, it's required that you set up the needed SQL table by using the SQL script listed on the camel-sql documentation web page.

JpaMessageIdRepository

`JpaMessageIDRepository` is similar to `JdbcMessageIdRepository`, but uses JPA as the SQL layer. This store is provided in the camel-jpa module. To use this store, JPA can automatically create needed SQL tables.

HazelcastIdempotentRepository

`HazelcastIdenpotentRepository` is a store using the Hazelcast in-memory data grid as a shared Map of the keys across the cluster of JVMs. Hazelcast can be configured to be in-memory only or to write the Map structure to persistent store by using different levels of QoS. This imple-

mentation is suitable for clustering. This store is provided in the camel-hazelcast module.

InfinispanIdempotentRepository

Similar to Hazelcast, `InfinispanIdempotentRepository` uses JBoss Infinispan instead as the in-memory data grid. This store is provided in the camel-infinispan module.

There are also idempotent repositories for Cassandra, HBase, MongoDB, and Redis that you can find in the corresponding Camel component. You can also write your own custom implementation by implementing `org.apache.camel.spi.IdempotentRepository`.

CACHE EVICTION

Each implementation of the idempotent repository has a different strategy for cache eviction. In-memory-based implementations have a limited size, with the number of entries they keep in the cache using a Least Recently Used (LRU) cache. The size of this cache can be configured. When there are more keys to be added to the cache, the least used is discarded first.

Other implementations such as SQL and JPA require you to manually delete unwanted records from the database table. In-memory data grids such as Hazelcast, Infinispan, and JCache provide configurations for setting the eviction strategy.

It's recommended to study the documentation of the technology you're using to ensure that you use the proper eviction strategy.

12.5.3 Clustered idempotent repository

This section covers a common use-case involving clustering and the Idempotent Consumer pattern. In this use-case, a shared filesystem is used for exchanging data that you want to process as fast as possible by scaling up your system in a cluster of active nodes; each node can pick up incoming files and process them.

The file component from Apache Camel has built-in support for idempotency. But for historical reasons, this implementation doesn't support atomic operations that a clustered solution would require; there's a small window of opportunity in which duplicates could still happen. Figure 12.19 illustrates this principle.

Figure 12.19 Two clustered nodes ❶ ❷ with the same Camel application compete concurrently to pick up new files from a shared filesystem ❸, and they may pick up the same file ❹, which causes duplicates.

When a new file is written to the shared filesystem 3, each active node 1 2 in the cluster reacts independently. Depending on timing and nondeterministic factors, the same file can potentially be picked up by one or more nodes 4. This isn't what you want to happen. You want only one node to process a given file at any time, but you also want each node to process different files at the same time, to achieve higher throughput and distribute the work across the nodes in the cluster, so node 1 can process file A while node 2 processes file B, and so forth.

What you need is something stronger. Yeah, maybe take a break. Grab a strong drink, or if you're in the office, a cup of coffee or tea.

The solution you're looking for is a clustered idempotent repository that the nodes use to coordinate and grant locks to exactly one node. Figure 12.20 shows how this plays out in Camel.

Figure 12.20 Two clustered nodes ❶❷ with the same Camel application compete concurrently to pick up new files from a shared filesystem ❸. Both nodes try to acquire a read lock ❹ that will permit exactly one with permission to pick up and process the file ❺. The other nodes will fail to acquire the lock and will skip attempting to pick up and process the file.

Each node in the cluster ❶ ❷, as before, reacts when new files are available from the shared filesystem ❸. When a node detects a file to pick up, it now first attempts to grab an exclusive read lock ❹ from the clustered idempotent repository. If granted, the file consumer can pick up and process the file ❺. And when it's done with the file, the read lock is released. While the read lock is granted to a node, any other node that attempts to acquire a read lock for the same filename would be denied by the clustered idempotent repository. The read lock is per file, so each node can process different files concurrently.

Let's see how to do this in Camel using the camel-hazelcast module.

Using Hazelcast as clustered idempotent repository

The camel-hazelcast module provides a clustered idempotent repository with the class name `HazelcastIdempotentRepository`. To use this repository, you first need to configure Hazelcast, which you can either use as a client to an existing external Hazelcast cluster or embed Hazelcast together with your Camel application.

Hazlecast can be configured by using Java code or from an XML file. This example uses the latter. We've copied the sample config that's distributed from Hazelcast and located the file in src/main/resources/hazelcast.xml. Hazelcast allows each node to autojoin using multicast. By doing this, we don't have to manually configure hostnames, IP addresses, and so on for each node to be part of the cluster. We had to change the hazelcast.xml configuration from using UDP to TCP with `localhost` to make the example run out of the box on personal computers, as not all popular operating systems have UDP enabled out of the box. We've left code comments in the hazelcast.xml file indicating how we did that.

The book's source code contains this example in the chapter12/hazelcast directory. To represent two nodes, we have two almost identical source files, camelinaction.ServerFoo and camelinaction.ServerBar. The following listing shows the source code for ServerFoo.

Listing 12.13 A standalone Java Camel application with Hazelcast embedded

```
public class ServerFoo {

  private Main main;

  public static void main(String[] args) throws Exception {
      ServerFoo foo = new ServerFoo();
    foo.boot(); (1)
```

(1) ServerFoo is a standalone Java application with a main method to boot the application.

```
    }

    public void boot() throws Exception {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();  ❷
```

❷ Creates and embeds a Hazelcast instance in server mode

```
      HazelcastIdempotentRepository repo =
        new HazelcastIdempotentRepository(hz, "camel");  ❸
```

❸ Sets up Camel Hazelcast idempotent repository using the name camel

```
      main = new Main();
      main.enableHangupSupport();
      main.bind("myRepo", repo);  ❹
```

❹ Binds the idempotent repository to the Camel registry with the name myRepo

```
      main.addRouteBuilder(new FileConsumerRoute("FOO", 100));  ❺
```

❺ Adds a Camel route from Java code and runs the application

```
      main.run();
    }
  }
```

Each node is a standalone Java application with a main method ❶. Hazelcast is configured using the XML file src/main/resources/hazelcast.xml. Then you embed and create a Hazelcast instance ❷, which by default reads its configuration from the root classpath as the name hazelcast.xml. At this point, Hazelcast is starting up and therefore is ready to use the Hazelcast idempotent repository created ❸. The remainder of the code uses Camel `Main` to easily configure (❹ ❺) and embed a Camel application with the route.

The Camel route that uses the idempotent repository is shown in the following listing.

Listing 12.14 Camel route using clustered idempotent repository with the file consumer

```
public class FileConsumerRoute extends RouteBuilder {
  public void configure() throws Exception {
      from("file:target/inbox" +
              "?delete=true" +
      "&readLock=idempotent" +    ❶
```

❶ The read lock must be configured as idempotent.

```
              "&readLockLoggingLevel=WARN" +
      "&idempotentRepository=#myRepo")    ❷
```

❷ The idempotent repository to use

```
              .log(name + " - Received file: ${file:name}")
      .delay(delay)    ❸
```

❸ Delays processing the file so we humans have a chance to see what happens

```
            .log(name + " - Done file:     ${file:name}")
            .to("file:target/outbox");
    }
}
```

The Camel route is a file consumer that picks up files from the target/inbox directory (the shared directory, in this example). The consumer is configured to use idempotent ❶ as its read lock. The idempotent repository is configured with the value `#myRepo` ❷, which is the name used in listing 12.13. When a file is being processed, it's logged and delayed for a little bit, so the application runs a bit slower and you can better see what happens.

You can try this example located in the chapter12/hazelcast directory by running the following Maven goals for separate shells so they run at the same time. (You can also run them from your IDE, as it's a standard Java main application. The IDE typically supports this by having a right-click menu to run Java applications):

```
mvn compile exec:java -Pfoo
mvn compile exec:java -Pbar
```

When you start the second application, Hazelcast should log the cluster state:

```
Members [2] {
     Member [localhost]:5701
     Member [localhost]:5702 this
}
```

Here you can see that there are two members in the cluster, linked by using TCP via ports 5701 and 5702.

If you copy a bunch of files to the target/inbox directory, each node will compete concurrently to pick up and process the files. When a node can't acquire an exclusive read lock, a `WARN` is logged:

```
WARN  tentRepositoryReadLockStrategy - Cannot acquire read lock. Will sk
WARN  tentRepositoryReadLockStrategy - Cannot acquire read lock. Will sk
```

If you look at the output from both consoles, you should see that the `WARN` from one node isn't a `WARN` from the other node, and vice versa. You can reduce the number of read-lock content by shuffling the files in random order. This can be done by configuring the file endpoint with `shuffle=true`. For example, when we tried this, processing 126 files went from 59 `WARN`s to only 3 when shuffle was turned on.

TIP You can also use the camel-infinispan module instead of Hazelcast, as it offers similar clustering caching capabilities.

One last thing that needs to be configured is the eviction strategy.

Configuring an eviction strategy

Hazelcast by default keeps each element in its distributed cache forever, but this often isn't what you want. For example, in the future you may want to be able to pick up files that have the same names as previously processed files. Or if you're using unique filenames, keeping old files in the cache is a waste of memory. To remove these old filenames from the cache, an eviction strategy must be configured. Hazelcast has many options for this, and in this example we've configured the files to be in the cache for 60 seconds using the time-to-live option.

```
<time-to-live-seconds>60</time-to-live-seconds>
```

Speaking of time, it's time to end this chapter.

12.6 Summary and best practices

Transactions play a crucial role when grouping distinct events together so that they act as a single, coherent, atomic event. In this chapter, we looked at how transactions work in Camel and discovered that Camel lets Spring orchestrate and manage transactions. Using Spring transactions, Camel lets you use an existing and proven transaction framework that works well with the most popular application servers, message brokers, and database systems.

Here are the best practices you should take away from this chapter:

- *Use transactions when appropriate*—Transactions can be used by only a limited number of resources, such as JMS and JDBC. Therefore, it makes sense to use transactions only when you can use these kinds of resources. If transactions aren't applicable, you can consider using your own code to compensate and to work as a rollback mechanism.
- *Local or global transactions*—If your route involves only one transactional resource, use local transactions. They're simpler and much less resource intensive. Use global transactions only if multiple resources are involved.
- *Configuring global transactions is difficult*—When using global transactions (JTA/XA), be advised that they're difficult to configure. The configuration is specific to the vendor and application server. Don't underestimate how difficult the configuration can be.
- *Short transactions*—Avoid building systems in which transactions span across a lot of business logic and are routed to many other systems. Keep your transactions short and simple. Instead of one long transaction, break it up into two or more individual steps and use message queues between them. Then you can use short transactions between those message queues and at the same time benefit from using queues that are a great way of decoupling producers and consumers.
- *Favor using one propagation scope*—Attempt to keep your transaction simple by using only one transactional propagation scope, to let all work be within the same transactional context.
- *Test your transactions*—Build unit and integration tests to ensure that your transactions work as expected.
- *Use synchronization tasks*—When you need custom logic to be executed as either a commit or rollback, use Camel's `Synchronization` as part of the exchange unit of work.
- *Use idempotent consumer*—To avoid processing the same message multiple times, use the Idempotent Consumer EIP pattern. To keep state, Camel offers different idempotent repository implementations. Make sure you understand and use the most appropriate for your use-case.
- *Clustered file consumer*—Make sure you use idempotent read lock with a clustered idempotent repository such as Hazelcast, Infnispan, or JCache when you want to scale out your application and have multiple Camel applications process files from a shared directory.

In chapter 13, we'll turn our attention to using parallel processing with Camel. You'll learn to how to improve performance, understand the threading model used in Camel, and more.