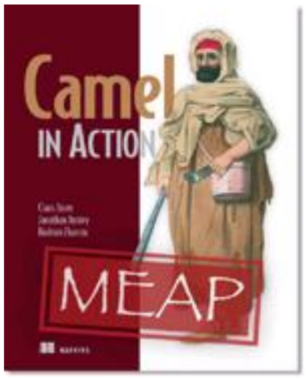
MANNING PUBLICATIONS

The role of transactions in Camel

Excerpted from



Camel in Action
EARLY ACCESS EDITION

Claus Ibsen, Jonathan Anstey, and Hadrian Zbarcea
MEAP Release: October 2009
Softbound print: Summer 2010 | 435 pages
ISBN: 9781935182368

This article is taken from the book Camel in Action. *The authors explain how transactions work in Camel and reveal that Camel lets Spring Transaction orchestrate and manage the transactions.*

To help explain what transaction is, we can give an example from a real life. Say you want to order *Camel in Action* from Manning Publications' online bookstore. To do that, you follow these steps in the specified order:

- Find the book Camel in Action
- Put the book in the basket
- Maybe look for other books and continue shopping
- Go to the checkout
- Fill out the shipping and credit card details
- Confirm the purchase
- Wait for the confirmation
- Leave the web store

What seems like an everyday scenario is actually a fairly complex orchestration of events. The events must happen in a specific order—you have to put books in the basket before you can check out. Before you can confirm the purchase, you must have filled out the required information. If your credit card is declined, the purchase will not be confirmed. The ultimate resolution of this transaction must be in one of two states: either the purchase was accepted and confirmed or the purchase was declined, leaving your credit card uncharged.

Okay, this real-life story does involve computer systems because the Manning bookstore is an online bookstore, but the story could just as well have been a shopping experience at a supermarket. You leave the supermarket either with or without your groceries.

In the software world, a transaction can be explained using a database as an example. A comparable series of events is SQL statements that manipulate database tables, such as updating or inserting data. While the transaction is in progress, a system failure could occur and that would leave the participants in an inconsistent state. That is why these series of events are described as atomic; either they all carry out or they all fail—it's all or nothing. In transactional terms, it's either commit or rollback. At this point, we are sure the reader knows about database ACID options and will not drag out this introduction further by explaining what Atomic, Consistent, Isolated, and Durable means.

To explain the reasons for using transactions, we go through a scenario at Rider Auto Parts, which demonstrates what happens if you didn't use them. After this, we will learn more about transaction and the fact it's Spring Transaction that orchestrates the transactions.

Why use transactions?

There are many good reasons why using transaction makes sense. However, before we learn more about using transactions with Camel, we will spend some time building up a case that shows what goes wrong when you don't use transactions. In the case, we also set up and use JMS and JDBC components because they are the primary components supporting transactions.

Rider Auto Parts partner integration

Rider Auto Parts is having trouble in its day-to-day business with its partners. Lately, there has been a dispute between Rider Auto Parts and a partner regarding whether or not a service meets the agreed terms also known as Service Level Agreement (SLA). When incidents occur, it's often a labor intensive task to investigate and remedy the incident. You have developed an application that periodically scans the partner's servers and reports performance and uptime metrics to a JMS queue. Rider Auto Parts already has an existing incident management application with a web user interface at disposal for the upper level management. What's missing is integrating the gathered metrics and populating them to the database used by the incident management application. Figure 1 illustrates the task you are facing.

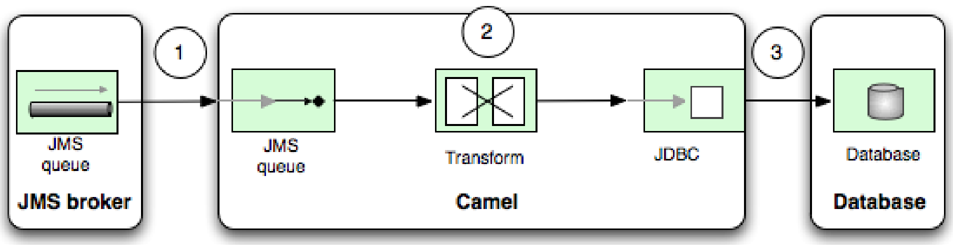


Figure 1 Partner reports is contumaciously received from the JMS broker, transformed in Camel to SQL format before it's written to the database

It's a fairly simple task because a JMS consumer is listening for new messages on the JMS queue (#1). Then the data is transformed (#2) from XML to SQL before it's written to the database (#3).

You have fairly good confidence that this is something you can do in less than an hour and you come up with a route that matches figure 1:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route id="partnerToDB">
        <from uri="activemq:queue:partners"/>
        <bean ref="partner" method="toSql"/>
        <to uri="jdbc:myDataSource"/>
    </route>
</camelContext>
```

The reports that are sent to the JMS queue are in a simple in-house XML format such as this example:

```
<?xml version="1.0"?>
<partner id="123">
    <date>200911150815</date>
    <code>200</code>
    <time>4387</time>
</partner>
```

The database table that should store 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) )
```

So that leaves you with a simple task of mapping the XML to the database. Because you are a pragmatic person and you 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 guns with JPA or Hibernate. So in short time you code a mapping code in a good old fashioned bean as shown in listing 1.

Listing 1 Using a bean to map from XML to SQL

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

public class PartnerServiceBean {

    public String toSql(@XPath("partner/@id") int id,                    #1
                        @XPath("partner/date/text()") String date,
                        @XPath("partner/code/text()") int statusCode,
                        @XPath("partner/time/text()") long responseTime) {

        StringBuilder sb = new StringBuilder();

        sb.append("INSERT INTO PARTNER_METRIC (partner_id, time_occurred,
                  status_code, perf_time) VALUES (");                    #2
        sb.append("'").append(id).append("', ");
        sb.append("'").append(date).append("', ");
        sb.append("'").append(statusCode).append("', ");
        sb.append("'").append(responseTime).append("') ");

        return sb.toString();
    }
}
```

#1 Using @XPath annotations to extract data from XML payload
#2 Constructing SQL statement

At this time, you can't believe that coding these 10 or so code lines was faster than it would have been just to get started on the JPA wagon or any heavyweight and proprietary mapping software. Okay, the code speaks for itself but let's go over it anyway. At first, we have defined the method to accept the four values to be mapped. Notice how we use the `@XPath` annotation to grab the data from the XML document (#1). Then, we use a `StringBuilder` to construct the `SQL INSERT` statement with the input values (#2).

Before showing this to anyone, you want to test it a bit and so you crank up a unit test.

```
public void testSendPartnerReportIntoDatabase() throws Exception {
    String sql = "select count(*) from partner_metric";
    assertEquals(0, jdbc.queryForInt(sql));                           #1

    String xml = "<?xml version=\"1.0\"?>
                + <partner id=\"123\"><date>200911150815</date>
                + <code>200</code><time>4387</time></partner>";

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

    Thread.sleep(5000);

    assertEquals(1, jdbc.queryForInt(sql));                           #2
}
```

#1 Asserts there are no rows in the database at start
#2 Asserts one row was inserted into the database

The test method outlines the principle. At first, you check that the database is empty and that it contains no rows (#1). Then, you construct a XML sample data, which you send to the JMS queue using Camel's `ProducerTemplate`. Since the processing of the JMS message is happening asynchronously, you must wait a bit to let it process. And, at the end, you check that the database now contains one row (#2).

Setting up the JMS broker and the database

To help run this unit test, you need to use a local JMS Broker and a database as well. For that, you use Apache ActiveMQ as JMS broker and HSQLDB as the database. HSQLDB is used as an in-memory database eliminating the need to run it separately. Apache ActiveMQ is an extremely versatile broker and even embeddable in unit tests. So, 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 listing 2.

Listing 2 XML configuration of Camel route, JMS broker, and database

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

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route id="partnerToDB">
        <from uri="activemq:queue:partners"/>
        <bean ref="partner" method="toSql"/>
        <to uri="jdbc:myDataSource"/>
    </route>
</camelContext>

<bean id="activemq"                                                    #1
      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>                                       #2
        <broker:transportConnector uri="tcp://localhost:61616"/>
    </broker:transportConnectors>
</broker:broker>

<bean id="myDataSource"                                                #3
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:mem:partner"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
</bean>
```

#1 Configuring the ActiveMQ component
#2 Setting up an embedded JMS broker
#3 Setting up the database

First, we define the partner bean from listing 1 as a spring bean, which we use in the route. To allow Camel to connect to ActiveMQ, we must define it as a Camel component (#1). The `brokerURL` property is configured with the URL for the remote ActiveMQ broker, which, in our example, happens to be running on same machine. Then, we set up a local embedded ActiveMQ broker (#2), which is configured to use TCP connectors. The last configuration needed is to set up the JDBC data source (#3).

TIP

If you use an embedded ActiveMQ, you can use the VM protocol instead of TCP, which bypasses the entire TCP stack and, therefore, is much faster. For example, in listing 2, we could have used `vm://localhost` instead of `tcp://localhost:61616`.

The story of the lost message

Now that you have configured all of the pieces needed, you run the unit test and it's all green. You share your knowledge with your team. Then it happens; Joe in the corner asks what happens if the connection to the database is failing. That's a great question, so you write another unit test that covers this situation.

You can simulate the connection failure using Camel interceptors. To write the unit test, you write all of the logic in a single method, as shown in listing 3.

Listing 3 Simulating connection failure which causes the lost message issue

```
public void testNoConnectionToDatabase() throws Exception {
    RouteBuilder rb = new RouteBuilder() {                              #1
        public void configure() throws Exception {
            interceptSendToEndpoint("jdbc:*")
                .throwException(new ConnectException("Cannot connect"));
        }
    };

    RouteDefinition route = context.getRouteDefinition("partnerToDB");
    route.adviceWith(rb);                                               #2

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

    String xml = "<?xml version=\"1.0\"?>
                 + <partner id=\"123\"><date>200911150815</date>
                 + <code>200</code><time>4387</time></partner>";

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

    Thread.sleep(5000);

    assertEquals(0, jdbc.queryForInt(sql));                             #3
}
```

#1 Simulating no connection to database
#2 Advices the simulation into the existing route
#3 Asserts no rows inserted into database

To test the failed connection to the database, we need to intercept the routing to the database and simulate the error. We do this using the `RouteBuilder`, where we have defined this scenario (#1). Now we need to advice the existing route with our interceptor (#2), which will automatic recreate the route and start it again. The remainder of the code is almost identical to the previous test although, at the end, we test that no rows were added to the database (#3).

Your team is impressed with unit test. But Joe has another question. What happens to the message; did it get lost?

After digging into this some more, you realize that the message is lost. But why? It's lost because we are *not* using transactions. What happens is that the JMS consumer is by default using auto acknowledge mode, which means the client acknowledges the message on receive and the message is dequeued from the JMS broker. What we must do is use transacted acknowledge mode instead.

At last, the purpose of transactions

With transactions, the start is often named *begin*, and the end *commit* (or *rollback* in case the transaction should undo). Figure 2 illustrates this principle.

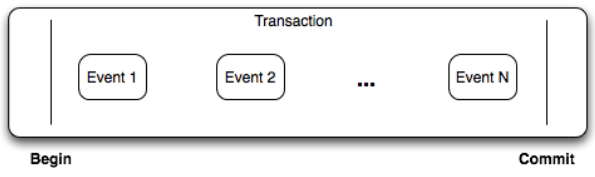


Figure 2 A transaction is a series of events between begin and commit

To help understand figure 2, you could write what is known as *locally managed transactions*, where the transaction is managed manually in the code. Following code, which is based on using JPA managed transactions, illustrates this point.

```
EntityManager em = emf.getEntityManager();
EntityTransaction tx = em.getTransaction();
try {
    tx.begin();


    tx.commit();
} catch (Exception e) {
    tx.rollback();
}
em.close();
```

The principle is that you start the transaction using the `begin` method. Then you do your series of work steps. And, at the end, you either commit or rollback the transaction depending on whether an exception was thrown or not.

Maybe you are already familiar with this principle but we go over this because, when we move on to learn how transactions work in Camel, the same principle is applied at a higher level of abstraction. This higher level of abstraction does not use locally managed transactions, which means that you are not invoking `begin` and `commit` methods from Java code. All this is abstracted at a higher level using declarative transactions. In other words, you do not program transaction in Java code but configure it in a Spring XML file.

In the next section, we'll see how this works so do not worry if it's a bit unclear at the moment.

Why declarative transactions?

So what are the benefits of using declarative notation? The vision from Spring is that you can configure all of this in Spring XML regardless of which kind of runtime environment you are using. It removes the need for changing any Java code to match the targeted environment. Another benefit is the fact that Spring makes it easy to set up diverse environments using minimal configuration effort. This technology is known as Spring Transaction[1]. It's a great piece of technology and the reason why Camel leverages it instead of rolling out its own transaction framework.

Now that we have established the fact Camel works together with Spring Transaction, let's talk a bit more about the principles of how they work together.

About Spring transaction

To understand how Camel works together with Spring Transaction we turn our attention to figure 3.

[1] http://static.springsource.org/spring/docs/2.5.x/reference/transaction.html

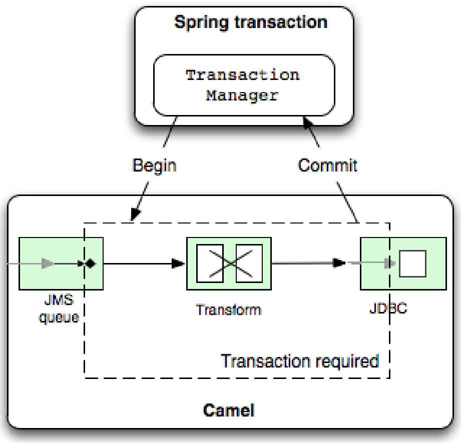


Figure 3 Spring TransactionManager is orchestrating the transaction by issuing begins and commits. In Camel, the entire route is transacted, which is handled by Spring Transaction.

Figure 3 shows that Spring transaction orchestrates a transaction, while Camel takes care of the rest. The figure is not entirely accurate because the JMS Broker does take active part in the transaction. Figure 4 includes the broker and shows a scenario we need to accomplish with the Rider Auto Parts partner application.

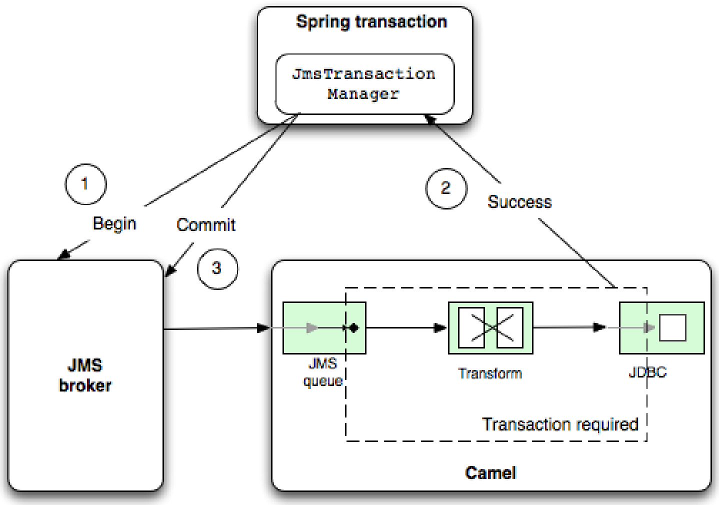


Figure 4 Spring JmsTransactionManager orchestrates the transaction by issuing begin to the JMS Broker. The Camel route successfully completes and JmsTransactionManager issues commit.

Figure 4 is a recap of figure 3 at a higher level where we can see the participating resources: JMS broker and Camel. First, `JmsTransactionManager` issues a begin (#1) to the transactional resource, which is the JMS broker. Depending on whether the Camel routes were completed with a success or failure (#2), `JmsTransactionManager` issues either a commit or rollback. In this example, we have shown the success scenario where a commit is issued (#3). Had the Camel route failed as a result of a thrown exception, the transaction manager would have issued a rollback instead.

Adding transactions to Auto Rider Parts partner example

We have just learned in the previous section that it's in fact Spring Transaction that manages transactions in Camel. In this section, you will learn how this applies in practice as we continue working on the use case.

Adding a transaction

Earlier in this article, we left the Rider Auto Parts use case with the problem of lost messages as a result of not using transactions. Your task now is to apply transactions, which should remedy the problem. So how do you do this? Well, we start by introducing Spring Transactions to our Spring XML file and adjusting the configuration accordingly. Listing 4 shows how this is done.

Listing 4 XML configuration using Spring Transaction

```
<bean id="activemq"
      class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="transacted" value="true"/>                          #1
    <property name="transactionManager" ref="txManager"/>
</bean>

<bean id="txManager"                                                    #2
      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"/>
</bean>
```

#1 Enabled transacted acknowledge mode
#2 Configures the Spring JmsTransactionManager

First, we turn on `transacted` on the ActiveMQ component (#1), which instructs it to use transacted acknowledge mode. Now, we need to refer to the transaction manager used. The transaction manager is a Spring `JmsTransactionManager` (#2), which manages transactions in JMS messaging. The transaction manager needs to know how to connect to the JMS broker, by which we mean connection factory. At the connection factory, we configure the `brokerURL` pointing at our JMS broker.

TIP

`JmsTransctionManager` has other options for configuring the transaction behavior, such as timeouts and strategies for rollback on commit failure. Consult the Spring documentation[2] for details.

So far, we have only reconfigured beans in the Spring XML file, which is mandatory when using Spring Transaction. In Camel itself, we have not yet configured anything related to transactions. Camel offers great convention over configuration in terms of transaction support, so all you have to do is add `<transacted/>` to the route, right after `<from>`, as highlighted below.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route id="partnerToDB">
        <from uri="activemq:queue:partners"/>
        <transacted/>
        <bean ref="partner" method="toSql"/>
        <to uri="jdbc:myDataSource"/>
    </route>
</camelContext>
```

[2] http://static.springsource.org/spring/docs/2.5.x/reference/transaction.html

When you specify `<transacted/>` in a route, Camel is told to use transaction for this particular route. What Camel does under the covers is look up the Spring TransactionManager and leverage it. This is the convention over configuration kicking in.

Using `transacted` in Java DSL is just as easy, as highlighted below.

```
from("activemq:queue:partners")
    .transacted()
    .beanRef("partner", "toSql")
    .to("jdbc:myDataSource");
```

The convention over configuration only applies when you only have a single Spring `TransactionManager` configured. In more complex scenarios with multiple transaction managers you have to do additional configuration to set up a transaction.

NOTE

`<transacted/>` must be added right after `<from/>` to ensure the entire route is transactional. The reason why this is not enforced in the DSL is that the DSL is loosely defined in a broad way, which makes it easy to maintain and develop Camel, with tradeoffs in a few spots.

This is basically all you need to do to configure transactions in Camel. So let's see if our configuration is correct by testing it.

Testing transactions

When you test Camel routes using transactions, it's very common to use real transactional resources. In other words, we test with live resources such as a real JMS broker and a database. The source code in our examples uses Apache ActiveMQ and HSQLDB as live resources. We have picked these because they can be easily downloaded using Apache Maven and are lightweight and embeddable, which makes them perfect for unit testing. There is no upfront work needed to install them as a separate JMS broker and database since they can easily be embedded and set up directly from within a unit test. To understand how this works with testing transactions, we return to the Auto Rider Parts example.

Last time we ran a unit test, we lost the message when there was no connection to the database. So let's try that unit test again, this time with transactional support. You can do this by running the following maven goal:

```
mvn compile test -Dtest=RiderAutoPartsPartnerTXTest.
```

By running the unit test, you will notice a lot of stacktraces printed on the console containing, `among` others, the following snippet:

```
2009-11-22 12:47:22,158 [enerContainer-1] ERROR EndpointMessageListener        -
java.net.ConnectException: Cannot connect to the database
org.apache.camel.spring.spi.TransactedRuntimeCamelException: java.net.ConnectException: Cannot
connect to the database
        at
org.apache.camel.spring.spi.TransactionErrorHandler.wrapTransactedRuntimeException(Transaction
ErrorHandler.java:173)
        at
org.apache.camel.spring.spi.TransactionErrorHandler$1.doInTransactionWithoutResult(Transaction
ErrorHandler.java:123)
        at
org.springframework.transaction.support.TransactionCallbackWithoutResult.doInTransaction(Trans
actionCallbackWithoutResult.java:33)
        at
org.springframework.transaction.support.TransactionTemplate.execute(TransactionTemplate.java:1
28)
        at
org.apache.camel.spring.spi.TransactionErrorHandler.process(TransactionErrorHandler.java:86)
```

We can see from the stacktrace that `EndpointMessageListener` has logged the exception at `ERROR` level, which indicates that the transaction is being rolled back. This happens because `EndpointMessageListener` is a `javax.jms.MessageListener`, which is being invoked when a new message arrived on the JMS destination, and it will rollback the transaction, if an exception was thrown.

So where is the message now? It should be on the JMS queue? So let's add a little code to our unit test, which checks whether the message is on the queue. Following code is added at the end of the unit test method we saw in listing 3:

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

With great confidence you run the unit test to ensure that the message was not lost. However, the unit test fails 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.RiderAutoPartsPartnerTXTest.testNoConnectionToDatabase(RiderAutoPartsPartnerTXTe
st.java:96)
```

You wonder what you have done wrong because the message is still lost. You are using a transaction and it has been configured correctly. Digging into the stacktraces, you discover that the message is always redelivered six times and then no further redeliver is conducted.

You grab *ActiveMQ in Action* by Manning Publications and spend some time digging for answers. What you learn from the book is the fact that ActiveMQ does the redelivery according to its default settings, which is to redeliver at most six times before giving up and moving the message to a dead letter queue.

What you have seen in action is the Dead Letter Channel EIP pattern. ActiveMQ also implements this pattern, which ensures the broker will not be doomed because of a poison message that cannot be processed. So, instead of looking at the message on the partner's queue, we should look for the message in the default ActiveMQ dead letter queue, which is named `ActiveMQ.DLQ`. You change the code accordingly, and the test passes:

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

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

You want to do one additional test that covers a situation where the connection to the database only fails at first try but works on a subsequent call. Listing 5 shows how you archived that.

Listing 5 Unit test how to simulate rollback at first try and a commit at the second try

```
public void testFailFirstTime() throws Exception {
    RouteBuilder rb = new RouteBuilder() {
        public void configure() throws Exception {
            interceptSendToEndpoint("jdbc:*")
                .choice()                                               #1
                    .when(header("JMSRedelivered").isEqualTo("false"))
                        .throwException(new ConnectException("Cannot connect to the
database"))
                .end();
        }
    };

    context.getRouteDefinition("partnerToDB").adviceWith(rb);

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

```
        String xml = "<?xml version=\"1.0\"?>
                     + <partner id=\"123\"><date>200911150815</date>
                     + <code>200</code><time>4387</time></partner>";
        template.sendBody("activemq:queue:partners", xml);

        Thread.sleep(5000);

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

        Object dlq = consumer.receiveBodyNoWait("activemq:queue:ActiveMQ.DLQ");
        assertNull("Should not be in the DLQ", dlq);                        #2
    }
```

The idea is to throw the `ConnectionException` the first time. We rely on the fact that any message consumed from a JMS destination has a set of standard JMS headers. From these, we use the `JMSRedelivered` header, which is a boolean indicating whether the JMS message is being redelivered or not. The interceptor logic is done in a Camel `RouteBuilder` so we have the full DSL at our disposal. We use the Content Based Router (#1) to check whether the message is being redelivered or not. The idea is to test the `JMSRedelivered` header and only throw the exception if it's `false`, which means it's the very first try. The rest of the unit test is to verify correct behavior so we check that the database is empty before sending the message to the JMS queue. We let the routing complete. After the completion, we check that the database has one row. Because we previously got tricked by the JMS Broker dead letter queue, we also check that it's empty (#2).

Summary

Transactions play a crucial role when grouping distinct events together, appearing as a single coherent atomic event. In this article, we dived into how transactions work in Camel and discovered that Camel lets Spring transaction orchestrate and manage the transactions. By leveraging Spring transaction, Camel lets you use an existing and proven transaction framework, which works together with the most popular application servers and message brokers.