Red Hat JBoss Fuse 6.3

Transaction Guide

Using transactions to make your routes roll back ready

Last Updated: 2018-07-16

Red Hat JBoss Fuse 6.3 Transaction Guide

Using transactions to make your routes roll back ready

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

This guide describes the basic concepts of transactions, how to select and implement a transaction manager, how to access data using Spring, the various ways to demarcate transactions, and JMS transaction semantics.

Table of Contents

CHAPTER 1. INTRODUCTION TO TRANSACTIONS

Abstract

This chapter defines some basic transaction concepts and explains how to generate and build a simple transactional JMS example in Apache Camel.

1.1. BASIC TRANSACTION CONCEPTS

What is a transaction?

The prototype of a transaction is an operation that conceptually consists of a single step (for example, transfer money from account A to account B), but must be implemented as a series of steps. Clearly, such operations are acutely vulnerable to system failures, because a crash is likely to leave some of the steps unfinished, leaving the system in an inconsistent state. For example, if you consider the operation of transferring money from account A to account B: if the system crashes *after* debiting account A, but *before* crediting account B, the net result is that some money disappears into thin air.

In order to make such an operation reliable, it must be implemented as a *transaction*. On close examination, it turns out that there are four key properties a transaction must have in order to guarantee reliable execution: these are the so-called ACID properties of a transaction.

ACID properties of a transaction

The *ACID* properties of a transaction are defined as follows:

- *Atomic*—a transaction is an all or nothing procedure; individual updates are assembled and either committed or aborted (rolled back) simultaneously when the transaction completes.
- *Consistent*—a transaction is a unit of work that takes a system from one consistent state to another.
- *Isolated*—while a transaction is executing, its partial results are hidden from other entities accessing the transaction.
- *Durable*—the results of a transaction are persistent.

Transaction clients

A *transaction client* is an API or object that enables you to initiate and end transactions. Typically, a transaction client exposes operations that enable you to *begin*, *commit*, or *roll back* a transaction. In the context of the Spring framework, the **PlatformTransactionManager** exposes a transaction client API.

Transaction demarcation

Transaction demarcation refers to the initiating and ending of transactions (where transactions can be ended either by being committed or rolled back). Demarcation can be effected either explicitly (for example, by calling a transaction client API) or implicitly (for example, whenever a message is polled from a transactional endpoint).

Resources

A *resource* is any component of a computer system that can undergo a persistent or permanent change. In practice, a resource is almost always a database or a service layered over a database (for example, a message service with persistence). Other kinds of resource are conceivable, however. For example, an Automated Teller Machine (ATM) is a kind of resource: once a customer has physically accepted cash from the machine, the transaction cannot be reversed.

Transaction manager

A *transaction manager* is responsible for coordinating transactions across one or more resources. In many cases, a transaction manager is built into a resource. For example, enterprise-level databases generally include a transaction manager that is capable of managing transactions involving that database. But for transactions involving *more* than one resource, it is normally necessary to employ an *external* transaction manager implementation.

Managing single or multiple resources

For transactions involving a *single* resource, the transaction manager built into the resource can generally be used. For transactions involving *multiple* resources, however, it is necessary to use an external transaction manager or a transaction processing (TP) monitor. In this case, the resources must be integrated with the transaction manager by registering their XA switches. There is also an important difference between the types of algorithm that are used for committing single-resource systems and multiple-resource systems, as follows:

- *1-phase commit*—suitable for single-resource systems, this protocol commits a transaction in a single step.

- *2-phase commit*—suitable for multiple-resource systems, this protocol commits a transaction in two steps. Including multiple resources in a transaction introduces an extra element of risk: there is the danger that a system failure might occur after some, but not all, of the resources have been committed. This would leave the system in an inconsistent state. The 2-phase commit protocol is designed to eliminate this risk, ensuring that the system can *always* be restored to a consistent state after it is restarted.

Transactions and threading

To understand transaction processing, it is crucial to appreciate the basic relationship between transactions and threads: *transactions are thread-specific.* That is, when a transaction is started, it is attached to a specific thread (technically, a *transaction context* object is created and associated with the current thread). From this point on (until the transaction ends), all of the activity in the thread occurs within this transaction scope. Conversely, activity in any other thread does *not* fall within this transaction's scope (although it might fall within the scope of some other transaction).

From this, we can draw a few simple conclusions:

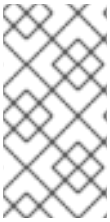
- *An application can process multiple transactions simultaneously*—as long as each of the transactions are created in separate threads.

- *Beware of creating subthreads within a transaction*—if you are in the middle of a transaction and you create a new pool of threads (for example, by calling the **threads()** DSL command), the new threads are *not* in the scope of the original transaction.

- *Beware of processing steps that* implicitly *create new threads*—for the same reason given in the preceding point.

- *Transaction scopes do not usually extend across route segments*—that is, if one route segment ends with **to(*JoinEndpoint*)** and another route segment starts with **from(*JoinEndpoint*)**, these

route segments typically do *not* belong to the same transaction. There are exceptions, however (see the section called “Breaking a route into fragments”).



NOTE

Some advanced transaction manager implementations give you the freedom to detach and attach transaction contexts to and from threads at will. For example, this makes it possible to move a transaction context from one thread to another thread. In some cases it is also possible to attach a single transaction context to multiple threads.

Transaction context

A *transaction context* is an object that encapsulates the information needed to keep track of a transaction. The format of a transaction context depends entirely on the relevant transaction manager implementation. At a minimum, the transaction context contains a unique transaction identifier.

Distributed transactions

A distributed transaction refers to a transaction in a distributed system, where the transaction scope spans multiple network nodes. A basic prerequisite for supporting distributed transactions is a network protocol that supports transmission of transaction contexts in a canonical format (see also, the section called “Distributed transaction managers”). *Distributed transaction lie outside the scope of Apache Camel transactions.*

X/Open XA standard

The X/Open XA standard describes a standardized interface for integrating resources with a transaction manager. If you want to manage a transaction that includes more than one resource, it is essential that the participating resources support the XA standard. Resources that support the XA standard expose a special object, the *XA switch*, which enables transaction managers (or TP monitors) to take control of their transactions. The XA standard supports both the 1-phase commit protocol and the 2-phase commit protocol.

1.2. TRANSACTION QUALITIES OF SERVICE

Overview

When it comes to choosing the products that implement your transaction system, there is a great variety of database products and transaction managers available, some free of charge and some commercial. All of them have nominal support for transaction processing, but there are considerable variations in the qualities of service supported by these products. This section provides a brief guide to the kind of features that you need to consider when comparing the reliability and sophistication of different transaction products.

Qualities of service provided by resources

The following features determine the quality of service of a resource:

- Transaction isolation levels.
- Support for the XA standard.

Transaction isolation levels

ANSI SQL defines four *transaction isolation levels*, as follows:

SERIALIZABLE

Transactions are perfectly isolated from each other. That is, nothing that one transaction does can affect any other transaction until the transaction is committed. This isolation level is described as *serializable*, because the effect is as if all transactions were executed one after the other (although in practice, the resource can often optimize the algorithm, so that some transactions are allowed to proceed simultaneously).

REPEATABLE_READ

Every time a transaction reads or updates the database, a read or write lock is obtained and held until the end of the transaction. This provides almost perfect isolation. But there is one case where isolation is not perfect. Consider a SQL **SELECT** statement that reads a range of rows using a **WHERE** clause. If another transaction adds a row to this range while the first transaction is running, the first transaction can see this new row, if it repeats the **SELECT** call (a *phantom read*).

READ_COMMITTED

Read locks are *not* held until the end of a transaction. So, repeated reads can give different answers (updates *committed* by other transactions are visible to an ongoing transaction).

READ_UNCOMMITTED

Neither read locks nor write locks are held until the end of a transaction. Hence, dirty reads are possible (that is, a transaction can see *uncommitted* updates made by other transactions).

Databases generally do not support all of the different transaction isolation levels. For example, some free databases support only **READ_UNCOMMITTED**. Also, some databases implement transaction isolation levels in ways that are subtly different from the ANSI standard. Isolation is a complicated issue, which involves trade offs with database performance (for example, see Isolation in Wikipedia).

Support for the XA standard

In order for a resource to participate in a transaction involving multiple resources, it needs to support the X/Open XA standard. You also need to check whether the resource's implementation of the XA standard is subject to any special restrictions. For example, some implementations of the XA standard are restricted to a single database connection (which implies that only one thread at a time can process a transaction involving that resource).

Qualities of service provided by transaction managers

The following features determine the quality of service of a transaction manager:

- Support for suspend/resume and attach/detach.
- Support for multiple resources.
- Distributed transactions.
- Transaction monitoring.
- Recovery from failure.

Support for multiple resources

A key differentiator for transaction managers is the ability to support multiple resources. This normally entails support for the XA standard, where the transaction manager provides a way for resources to register their XA switches.

NOTE

Strictly speaking, the XA standard is not the only approach you can use to support multiple resources, but it is the most practical one. The alternative typically involves writing tedious (and critical) custom code to implement the algorithms normally provided by an XA switch.

Support for suspend/resume and attach/detach

Some transaction managers support advanced capabilities for manipulating the associations between a transaction context and application threads, as follows:

- *Suspend/resume current transaction*—enables you to suspend temporarily the current transaction context, while the application does some non-transactional work in the current thread.
- *Attach/detach transaction context*—enables you to move a transaction context from one thread to another or to extend a transaction scope to include multiple threads.

Distributed transactions

Some transaction managers have the capability to manage transactions whose scope includes multiple nodes in a distributed system (where the transaction context is propagated from node to node using special protocols such as WS-AtomicTransactions or CORBA OTS).

Transaction monitoring

Advanced transaction managers typically provide visual tools to monitor the status of pending transactions. This kind of tool is particularly useful after a system failure, where it can help to identify and resolve transactions that were left in an uncertain state (heuristic exceptions).

Recovery from failure

There are significant variations amongst transaction managers with respect to their robustness in the event of a system failure (crash). The key strategy that transaction managers use is to write data to a persistent log before performing each step of a transaction. In the event of a failure, the data in the log can be used to recover the transaction. Some transaction managers implement this strategy more carefully than others. For example, a high-end transaction manager would typically duplicate the persistent transaction log and allow each of the logs to be stored on separate host machines.

1.3. GETTING STARTED WITH TRANSACTIONS

1.3.1. Prerequisites

Overview

The following are required to complete this example:

- Internet connection (required by Maven)

- Java Runtime
- Apache Maven 3

Java Runtime

Apache Camel requires a Java 7 development kit (JDK 1.7.0). After installing the JDK, set your **JAVA_HOME** environment variable to point to the root directory of your JDK, and set your **PATH** environment variable to include the Java **bin** directory.

Apache Maven 3

The Apache Camel Maven tooling requires Apache Maven version 3. To download Apache Maven, go to http://maven.apache.org/download.cgi.

After installing Apache Maven do the following:

1. Set your **M2_HOME** environment variable to point to the Maven root directory.
2. Set your **MAVEN_OPTS** environment variable to **-Xmx512M** to increase the memory available for Maven builds.
3. Set your **PATH** environment variable to include the Maven **bin** directory:

Platform	Path
Windows	**%M2_HOME%\bin**
UNIX	**$M2_HOME/bin**

1.3.2. Generate a New Project

Overview

Use the Maven archetype, **karaf-camel-cbr-archetype**, to generate a sample Java application which you can then use as a starting point for your application.

Steps

To generate the new project, perform the following steps:

1. Open a new command window and change to the directory where you want to store the new Maven project.
2. Enter the following command to generate the new Maven project:

```
mvn archetype:generate
 -DarchetypeGroupId=io.fabric8.archetypes
 -DarchetypeArtifactId=karaf-camel-cbr-archetype
 -DarchetypeVersion=1.2.0.redhat-630187
 -DgroupId=tutorial
```

```
-DartifactId=tx-jms-router
-Dversion=1.0-SNAPSHOT
-Dfabric8-profile=tx-jms-router-profile
```

Each time you are prompted for input, press Enter to accept the default.

This command generates a basic router application under the **tx-jms-router** directory. You will customize this basic application to demonstrate transactions in Apache Camel.

NOTE

Maven accesses the Internet to download JARs and stores them in its local repository.

3. Add dependencies on the artifacts that implement Spring transactions. Look for the **dependencies** element in the POM file and add the following **dependency** elements:

```
<project ...>
   ...
   <dependencies>
    ...
    <!-- Spring transaction dependencies -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-aop</artifactId>
    </dependency>

  </dependencies>
  ...
</project>
```

NOTE

It is not necessary to specify the versions of these artifacts, because this POM is configured to use the Fabric8 BOM, which configures default artifact versions through Maven's dependency management mechanism.

4. Add the JMS and ActiveMQ dependencies. Look for the **dependencies** element in the POM file and add the following **dependency** elements:

```
<project ...>
   ...
```

```
    <dependencies>
      ...
      <!-- Persistence artifacts -->
      <dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-jms</artifactId>
      </dependency>
      <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-client</artifactId>
      </dependency>

    </dependencies>
    ...
</project>
```

1.3.3. Configure a Transaction Manager and a Camel Route

Overview

The basic requirements for writing a transactional application in Spring are a *transaction manager bean* and a *resource bean* (or, in some cases, multiple resource beans). You can then use the transaction manager bean either to create a transactional Apache Camel component (see Section 5.2, “Demarcation by Transactional Endpoints”) or to mark a route as transactional, using the **transacted()** Java DSL command (see Section 5.1, “Demarcation by Marking the Route”).

Steps

To configure a JMS transaction manager and a Camel route in Blueprint XML, perform the following steps:

1. Customize the Blueprint XML configuration. Using your favourite text editor, open the **tx-jms-router/src/main/resources/OSGI-INF/blueprint/cbr.xml** file and replace the contents of the file with the following XML code:

```
<?xml version="1.0"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
          http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
          http://camel.apache.org/schema/blueprint
http://camel.apache.org/schema/blueprint/camel-blueprint.xsd">

   <camelContext xmlns="http://camel.apache.org/schema/blueprint"
xmlns:order="http://fabric8.com/examples/order/v7"
      id="tx-jms-router-context">
    <route>
      <from uri="file:work/data?noop=true"/>
      <convertBodyTo type="java.lang.String"/>
      <to uri="jmstx:queue:giro"/>
    </route>
    <route>
      <from uri="jmstx:queue:giro"/>
```

```
      <to uri="jmstx:queue:credits"/>
      <to uri="jmstx:queue:debits"/>
      <bean ref="myTransform" method="transform"/>
    </route>
  </camelContext>

  <bean id="myTransform" class="tutorial.MyTransform"/>

  <bean id="jmstx" class="org.apache.camel.component.jms.JmsComponent">
    <property name="configuration" ref="jmsConfig" />
  </bean>

  <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
     <property name="connectionFactory" ref="jmsConnectionFactory"/>
     <property name="transactionManager" ref="jmsTransactionManager"/>
     <property name="transacted" value="true"/>
  </bean>

  <bean id="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"/>
    <property name="userName" value="Username"/>
    <property name="password" value="Password"/>
  </bean>

</blueprint>
```

2. In the **jmsConnectionFactory** bean from the preceding Spring XML code, customize the values of the **userName** and **password** property settings with one of the user credentials from the JBoss Fuse container. By default, the container's user credentials are normally defined in the **etc/users.properties** file.

1.3.4. Create the MyTransform Bean

Overview

The purpose of the **MyTransform** bean class is to force a rollback of the current transaction, by throwing an exception. The bean gets called at the end of the second transactional route. This enables you to verify the behaviour of a rolled back transaction.

Steps

Create the **MyTransform** bean class. Using your favourite text editor, create the **tx-jms-router/src/main/java/tutorial/MyTransform.java** file and add the following Java code to the file:

```
package tutorial;

import java.util.Date;
import java.util.logging.Logger;
```

```
public class MyTransform {
   private static final transient Logger LOGGER = Logger.getLogger(MyTransform.class.getName());

   public String transform(String body)
   throws java.lang.Exception
   {
      // should be printed n times due to redeliveries
      LOGGER.info("message body = " + body);
      // force rollback
      throw new java.lang.Exception("test");
   }

}
```

1.3.5. Build and Run the Example

Overview

After building and running the example using Maven, you can use the Fuse Management Console to examine what has happened to the JMS queues involved in the application.

Steps

To build and run the transactional JMS example, perform the following steps:

1. To build the example, open a command prompt, change directory to **tx-jms-router**, and enter the following Maven command:

   ```
   mvn install
   ```

   If the build is successful, you should see the file, **tx-jms-router.jar**, appear under the **tx-jms-router**/**target** directory.

2. Create a sample message for the routes to consume when they are running in the container. Create the following directory path in the container's installation directory (where you installed JBoss Fuse):

   ```
   InstallDir/work/data
   ```

   In the **data** directory create the file, **message.txt**, with the following contents:

   ```
   Test message.
   ```

3. Start up the JBoss Fuse container. Open a new command prompt and enter the following commands:

   ```
   cd InstallDir/bin
   ./fuse
   ```

4. To install and start the example in the container, enter the following console command:

   ```
   JBossFuse:karaf@root> install -s mvn:tutorial/tx-jms-router/1.0-SNAPSHOT
   ```

5. To see the result of running the routes, open the container log using the **log:display** command, as follows:

```
JBossFuse:karaf@root> log:display
```

   If all goes well, you should see about a dozen occurrences of **java.lang.Exception: test** in the log. This is the expected behaviour.

6. What happened? The series of runtime exceptions thrown by the application is *exactly* what we expect to happen, because the route is programmed to throw an exception every time an exchange is processed by the route. The purpose of throwing the exception is to trigger a transaction rollback, causing the current exchange to be un-enqueued from the **queue:credit** and **queue:debit** queues.

7. To gain a better insight into what occurred, user your browser to connect to the Fuse Management Console. Navigate to the following URL in your browser:

```
http://localhost:8181/hawtio
```

   You will be prompted to log in. Use one of the credentials configured for your container (usually defined in the ***InstallDir*/etc/users.properties** file).

8. Click on the **ActiveMQ** tab to explore the JMS queues that are accessed by the example routes.

9. Drill down to the **giro** queue. Notice that the **EnqueueCount** and **DequeueCount** for **giro** are all equal to 1, which indicates that one message entered the queue and one message was pulled off the queue.

10. Click on the **debits** queue. Notice that the **EnqueueCount**, **DispatchCount**, and **DequeueCount** for **debits** are all equal to 0. This is because the **test** exception caused the enqueued message to be rolled back each time an exchange passed through the route. The same thing happened to the **credits** queue.

11. Click on the **ActiveMQ.DLQ** queue. The **DLQ** part of this name stands for *Dead Letter Queue* and it is an integral part of the way ActiveMQ deals with failed message dispatches. In summary, the default behavior of ActiveMQ when it fails to dispatch a message (that is, when an exception reaches the JMS consumer endpoint, **jmstx:queue:giro**), is as follows:

    a. The consumer endpoint attempts to redeliver the message. Redelivery attempts can be repeated up to a configurable maximum number of times.

    b. If the redeliveries limit is exceeded, the consumer endpoint gives up trying to deliver the message and enqueues it on the dead letter queue instead (by default, **ActiveMQ.DLQ**).

    You can see from the status of the **ActiveMQ.DLQ** queue that the number of enqueued messages, **EnqueueCount**, is equal to 1. This is where the failed message has ended up.

CHAPTER 2. SELECTING A TRANSACTION MANAGER

Abstract

This chapter describes how to select and configure a transaction manager instance in Spring. Most of the difficult work of configuring transactions consists of setting up the transaction manager correctly. Once you have completed this step, it is relatively easy to use transactions in your Apache Camel routes.

2.1. WHAT IS A TRANSACTION MANAGER?

Transaction managers in Spring

A *transaction manager* is the part of an application that is responsible for coordinating transactions across one or more resources. In the Spring framework, the transaction manager is effectively the root of the transaction system. Hence, if you want to enable transactions on a component in Spring, you typically create a transaction manager bean and pass it to the component.

The responsibilities of the transaction manager are as follows:

- *Demarcation*—starting and ending transactions using *begin*, *commit*, and *rollback* methods.
- *Managing the transaction context*—a transaction context contains the information that a transaction manager needs to keep track of a transaction. The transaction manager is responsible for creating transaction contexts and attaching them to the current thread.
- *Coordinating the transaction across multiple resources*—enterprise-level transaction managers typically have the capability to coordinate a transaction across multiple resources. This feature requires the 2-phase commit protocol and resources must be registered and managed using the XA protocol (see the section called “X/Open XA standard”).

  This is an advanced feature, not supported by all transaction managers.
- *Recovery from failure*—transaction managers are responsible for ensuring that resources are not left in an inconsistent state, if there is a system failure and the application crashes. In some cases, manual intervention might be required to restore the system to a consistent state.

Local transaction managers

A *local transaction manager* is a transaction manager that can coordinate transactions over a *single* resource only. In this case, the implementation of the transaction manager is typically embedded in the resource itself and the Spring transaction manager is just a thin wrapper around this built-in transaction manager.

For example, the Oracle database has a built-in transaction manager that supports demarcation operations (using SQL operations, **BEGIN**, **COMMIT**, **ROLLBACK**, or using a native Oracle API) and various levels of transaction isolation. Control over the Oracle transaction manager can be exported through JDBC, which is how Spring is able to wrap this transaction manager.

It is important to understand what constitutes a *resource*, in this context. For example, if you are using a JMS product, the JMS resource is the single running instance of the JMS product, *not* the individual queues and topics. Moreover, sometimes, what appears to be multiple resources might actually be a single resource, if the same underlying resource is accessed in different ways. For example, your

application might access a relational database both directly (through JDBC) and indirectly (through an object-relational mapping tool like Hibernate). In this case, the same underlying transaction manager is involved, so it should be possible to enrol both of these code fragments in the same transaction.

NOTE

It cannot be guaranteed that this will work in every case. Although it is possible in principle, some detail in design of the Spring framework or other wrapper layers might prevent it from working in practice.

Of course, it is possible for an application to have many different local transaction managers working independently of each other. For example, you could have one route that manipulates JMS queues and topics, where the JMS endpoints reference a JMS transaction manager. Another route could access a relational database through JDBC. But you could not combine JDBC and JMS access in the same route and have them both participate in the same transaction.

Global transaction managers

A *global transaction manager* is a transaction manager that can coordinate transactions over *multiple* resources. In this case, you cannot rely on the transaction manager built into the resource itself. Instead, you require an external system, sometimes called a *transaction processing monitor* (TP monitor), that is capable of coordinating transactions across different resources.

The following are the prerequisites for global transactions:

- *Global transaction manager or TP monitor*—an external transaction system that implements the 2-phase commit protocol for coordinating multiple XA resources.
- *Resources that support the XA standard*—in order to participate in a 2-phase commit, resources must support the X/Open XA standard. In practice, this means that the resource is capable of exporting an *XA switch* object, which gives complete control of transactions to the external TP monitor.

TIP

The Spring framework does *not* by itself provide a TP monitor to manage global transactions. It does, however, provide support for integrating with an OSGi-provided TP monitor or with a J2EE-provided TP monitor (where the integration is implemented by the **JtaTransactionManager** class). Hence, if you deploy your application into an OSGi container with full transaction support, you can use multiple transactional resources in Spring.

Distributed transaction managers

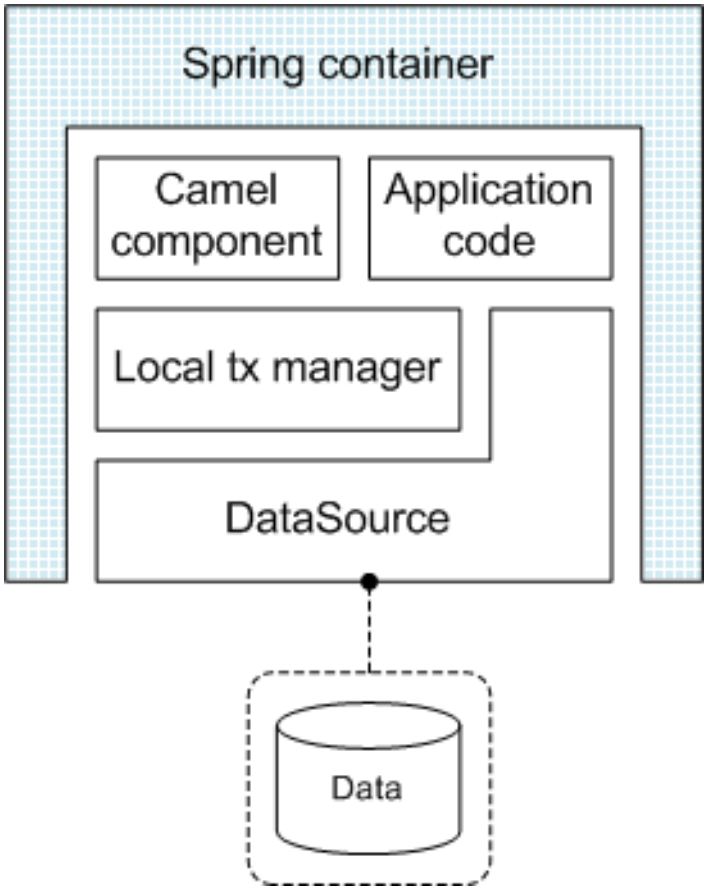
Usually, a server connects *directly* to the resources involved in a transaction. In a distributed system, however, it is occasionally necessary to connect to resources that are exposed only *indirectly*, through a Web service. In this case, you require a TP monitor that is capable of supporting *distributed transactions*. Several standards are available that describe how to support transactions for various distributed protocols—for example, the WS-AtomicTransactions specification for Web services.

2.2. SPRING TRANSACTION ARCHITECTURE

Overview

Figure 2.1, “Spring Transaction Architecture” shows an overview of the Spring transaction architecture.

Figure 2.1. Spring Transaction Architecture



Standalone Spring container

In the standalone deployment model, the Spring container provides access to persistent data sources and is responsible for managing the transactions associated with those data sources. A notable limitation of the standalone model is that the Spring container can support only *local* transaction managers, which means that only one data source (resource) at a time can participate in a transaction.

Data source

Spring supports a variety of different wrapper APIs for accessing persistent storage. For example, to access a database through JDBC, Spring provides the **SimpleDriverDataSource** class to represent the database instance and the **JdbcTemplate** class to provide access to the database using SQL. Wrappers are also provided for other kinds of persistent resource, such as JMS, Hibernate, and so on. The Spring data sources are designed to be compatible with the local transaction manager classes.

Local transaction manager

In Spring, a local transaction manager is a wrapper class that is responsible for managing the transactions of a *single resource*. The local transaction manager is responsible for starting, committing, and rolling back transactions. Typically, the way that you use a transaction manager in Apache Camel is that you pass the transaction manager reference to a transactional Apache Camel component bean.

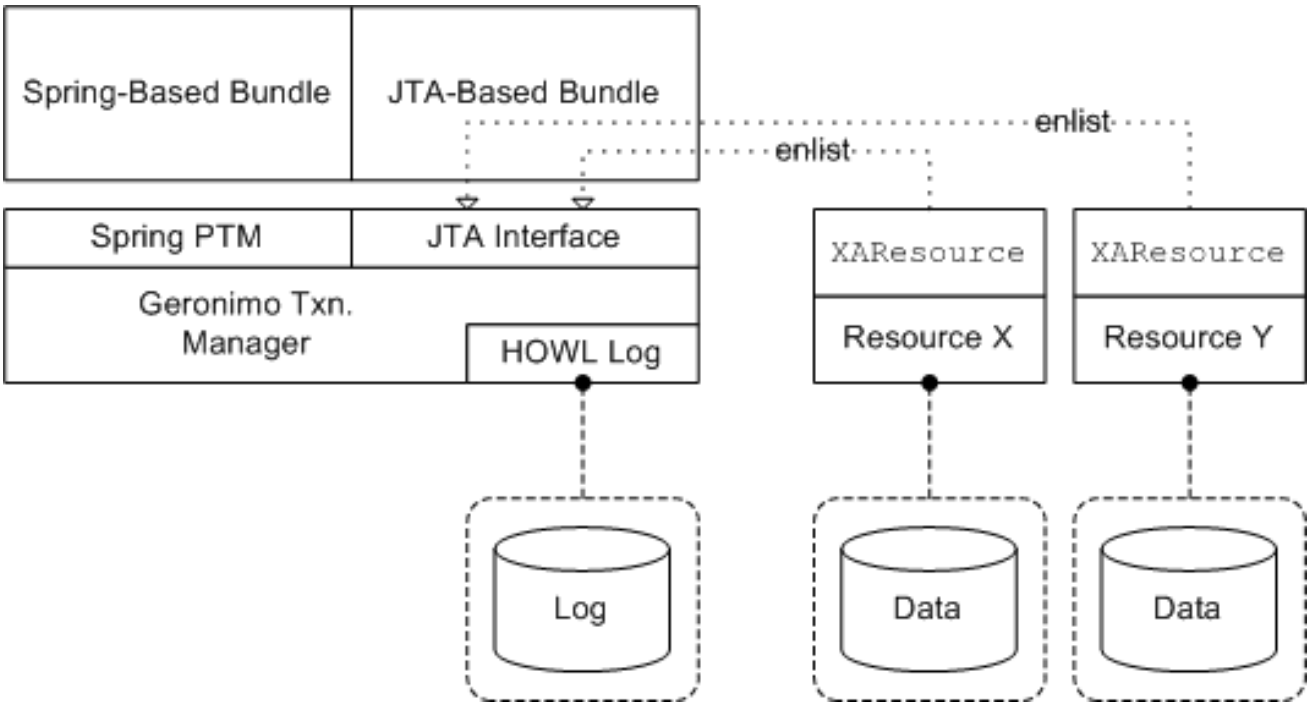
Spring provides different types of local transaction manager for different types of data source. For example, Spring provides a **DataSourceTransactionManager** for JDBC, a **JmsTransactionManager** for JMS, a **HibernateTransactionManager** for Hibernate, and so on.

2.3. OSGI TRANSACTION ARCHITECTURE

Overview

Figure 2.2, “OSGi Transaction Architecture” shows an overview of the OSGi transaction architecture in Red Hat JBoss Fuse. The core of the architecture is a JTA transaction manager based on Apache Geronimo, which exposes various transaction interfaces as OSGi services.

Figure 2.2. OSGi Transaction Architecture



OSGi mandated transaction architecture

The *JTA Transaction Services Specification* section of the *OSGi Service Platform Enterprise Specification* describes the kind of transaction support that can (optionally) be provided by an OSGi container. Essentially, OSGi mandates that the transaction service is accessed through the Java Transaction API (JTA).

The transaction service exports the following JTA interfaces as OSGi services (the *JTA services*):

- **javax.transaction.UserTransaction**
- **javax.transaction.TransactionManager**
- **javax.transaction.TransactionSynchronizationRegistry**

Only one *JTA provider* should be made available in an OSGi container. In other words, the JTA services are registered only once and the objects obtained by importing references to the JTA services must be unique.

Spring transaction integration

The Red Hat JBoss Fuse transaction service exports the following additional interfaces as OSGi services:

- **org.springframework.transaction.PlatformTransactionManager**

- **org.apache.geronimo.transaction.manager.RecoverableTransactionManager**

By obtaining a reference to the **PlatformTransactionManager** OSGi service, it is possible to integrate application bundles written using the Spring transaction API into the Red Hat JBoss Fuse transaction architecture.

Reference

For more details about the OSGi transaction architecture, see the introductory chapter from *Deploying into the OSGi Container*.

2.4. PLATFORMTRANSACTIONMANAGER INTERFACE

Overview

The **PlatformTransactionManager** interface is the key abstraction in the Spring transaction API, providing the classic transaction client operations: *begin*, *commit* and *rollback*. This interface thus provides the essential methods for controlling transactions at run time.

NOTE

The other key aspect of any transaction system is the API for implementing transactional resources. But transactional resources are generally implemented by the underlying database, so this aspect of transactional programming is rarely a concern for the application programmer.

PlatformTransactionManager interface

Example 2.1, “The PlatformTransactionManager Interface” shows the definition of the **org.springframework.transaction.PlatformTransactionManager** interface.

Example 2.1. The PlatformTransactionManager Interface

```
package org.springframework.transaction;

public interface PlatformTransactionManager {
  TransactionStatus getTransaction(TransactionDefinition definition)
    throws TransactionException;

  void commit(TransactionStatus status) throws TransactionException;

  void rollback(TransactionStatus status) throws TransactionException;
}
```

TransactionDefinition interface

The **TransactionDefinition** interface is used to specify the characteristics of a newly created transaction. It is used to specify the *isolation level* and the *propagation policy* of the new transaction. For more details, see Section 5.3, “Propagation Policies”.

TransactionStatus interface

The **TransactionStatus** interface can be used to check the status of the current transaction (that is, the transaction associated with the current thread) and to mark the current transaction for rollback. It is defined as follows:

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();

    boolean hasSavepoint();

    void setRollbackOnly();

    boolean isRollbackOnly();

    void flush();

    boolean isCompleted();
}
```

Using the PlatformTransactionManager interface

The **PlatformTransactionManager** interface defines the following methods:

getTransaction()

Create a new transaction and *associate it with the current thread*, passing in a **TransactionDefinition** object to define the characteristics of the new transaction. This is analogous to the *begin()* method of many other transaction client APIs.

commit()

Commit the current transaction, making permanent all of the pending changes to the registered resources.

rollback()

Roll back the current transaction, undoing all of the pending changes to the registered resources.

Generally, you do *not* use the **PlatformTransactionManager** interface directly. In Apache Camel, you typically use a transaction manager as follows:

1. Create an instance of a transaction manager (there are several different implementations available in Spring—see Section 2.5, “Transaction Manager Implementations”).
2. Pass the transaction manager instance either to a Apache Camel component or to the **transacted()** DSL command in a route. The transactional component or the **transacted()** command is then responsible for demarcating transactions (see Chapter 5, *Transaction Demarcation*).

2.5. TRANSACTION MANAGER IMPLEMENTATIONS

Overview

This section provides a brief overview of all the transaction manager implementations provided by the Spring framework. In general, the implementations fall into two different categories: *local transaction managers* and *global transaction managers*.

Local transaction managers

Table 2.1, “Local Transaction Managers” summarizes the local transaction manager implementations provided by the Spring framework. These transaction managers are distinguished by the fact that they support a *single resource only*.

Table 2.1. Local Transaction Managers

Transaction Manager	Description
JmsTransactionManager	A transaction manager implementation that is capable of managing a *single* JMS resource. That is, you can connect to any number of queues or topics, but *only* if they belong to the same underlying JMS messaging product instance. Moreover, you cannot enlist any other types of resource in a transaction. For example, using this transaction manager, it would *not* be possible to enlist both a SonicMQ resource and an Apache ActiveMQ resource in the same transaction. But see Table 2.2, “Global Transaction Managers”.
DataSourceTransactionManager	A transaction manager implementation that is capable of managing a *single* JDBC database resource. That is, you can update any number of different database tables, but *only* if they belong to the same underlying database instance.
HibernateTransactionManager	A transaction manager implementation that is capable of managing a Hibernate resource. It is not possible, however, to simultaneously enlist any other kind of resource in a transaction.
JdoTransactionManager	A transaction manager implementation that is capable of managing a Java Data Objects (JDO) resource. It is not possible, however, to simultaneously enlist any other kind of resource in a transaction.
JpaTransactionManager	A transaction manager implementation that is capable of managing a Java Persistence API (JPA) resource. It is not possible, however, to simultaneously enlist any other kind of resource in a transaction.
CciLocalTransactionManager	A transaction manager implementation that is capable of managing a Java Connection Architecture (JCA) resource. It is not possible, however, to simultaneously enlist any other kind of resource in a transaction.

Global transaction managers

Table 2.2, “Global Transaction Managers” summarizes the global transaction manager implementations provided by the Spring framework. These transaction managers are distinguished by the fact that they can support *multiple resources*.

Table 2.2. Global Transaction Managers

Transaction Manager	Description
JtaTransactionManager	If you require a transaction manager that is capable of enlisting more than one resource in a transaction, use the JTA transaction manager, which is capable of supporting the XA transaction API. You *must* deploy your application inside either an OSGi container or a J2EE server to use this transaction manager.
OC4JJtaTransactionManagner	A specialization of the **JtaTransactionManager** to work with Oracle's OC4J. The advantage of this implementation is that it makes Spring-driven transactions visible in OC4J's transaction monitor
WebLogicJtaTransactionManager	A specialization of the **JtaTransactionManager** to work with the BEA WebLogic container. Makes certain advanced transaction features available: transaction names, per-transaction isolation levels, and proper suspension/resumption of transactions.
WebSphereUowTransactionManager	A specialization of the **JtaTransactionManager** to work with the IBM WebSphere container. Enables proper suspension/resumption of transactions.

2.6. SAMPLE CONFIGURATIONS

2.6.1. JDBC Data Source

Overview

If you need to access a database, the JDBC data source provides a convenient, general-purpose mechanism for connecting to a database and making SQL based queries and updates. To group multiple updates into a single transaction, you can instantiate a Spring **DataSourceTransactionManager** and create a transaction scope using the **transacted()** DSL command.

Sample JDBC configuration

Example 2.2, “Data Source Transaction Manager Configuration” shows how to instantiate a JDBC transaction manager, of **DataSourceTransactionManager** type, which is required if you want to integrate a JDBC connection with Spring transactions. The JDBC transaction manager requires a reference to data source bean (created here with the ID, **dataSource**).

Example 2.2. Data Source Transaction Manager Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
      http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">
   ...
   <!-- spring transaction manager -->
   <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
      <property name="dataSource" ref="dataSource"/>
   </bean>

   <!-- datasource to the database -->
   <bean id="dataSource"
class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
      <property name="driverClass" value="org.hsqldb.jdbcDriver"/>
      <property name="url" value="jdbc:hsqldb:mem:camel"/>
      <property name="username" value="sa"/>
      <property name="password" value=""/>
   </bean>

</beans>
```

JDBC data source transaction manager bean

In Example 2.2, “Data Source Transaction Manager Configuration”, the **txManager** bean is a local JDBC transaction manager instance, of **DataSourceTransactionManager** type. There is just one property you need to provide to the JDBC transaction manager: a reference to a *JDBC data source*.

JDBC data source bean

In Example 2.2, “Data Source Transaction Manager Configuration”, the **dataSource** bean is an instance of a JDBC data source, of **javax.sql.DataSource** type. The JDBC data source is a standard feature of the Java DataBase Connectivity (JDBC) specification and it represents a single JDBC connection, which encapsulating the information required to connect to a specific database.

In Spring, the recommended way to create a data source is to instantiate a **SimpleDriverDataSource** bean (which implements the **javax.sql.DataSource** interface). The simple driver data source bean creates a new data source using a JDBC driver class (which is effectively a data source factory). The properties that you supply to the driver manager data source bean are specific to the database you want to connect to. In general, you need to supply the following properties:

driverClass

An instance of **java.sql.Driver**, which is the JDBC driver implemented by the database you want to connect to. Consult the third-party database documentation for the name of this driver class (some examples are given in Table 2.6, “Connection Details for Various Databases”).

url

The JDBC URL that is used to open a connection to the database. Consult the third-party database documentation for details of the URL format (some examples are given in Table 2.6, “Connection Details for Various Databases”).

For example, the URL provided to the **dataSource** bean in Example 2.2, “Data Source Transaction Manager Configuration” is in a format prescribed by the HSQLDB database. The URL, **jdbc:hsqldb:mem:camel**, can be parsed as follows:

- The prefix, **jdbc:hsqldb:**, is common to all HSQLDB JDBC connection URLs;
- The prefix, **mem:**, signifies an in-memory (non-persistent) database;
- The final identifier, **camel**, is an arbitrary name that identifies the in-memory database instance.

username

The username that is used to log on to the database.

For example, when a new HSQLDB database instance is created, the **sa** user is created by default (with administrator privileges).

password

The password that matches the specified username.

Standalone data sources

Spring provides a variety of data source implementations, which are suitable for standalone mode (that is, the application is *not* deployed inside an OSGi container). These data sources are described in Table 2.3, “Standalone Data Source Classes”.

Table 2.3. Standalone Data Source Classes

Data Source Class	Description
SimpleDriverDataSource	This data source should always be used in standalone mode. You configure this data source by providing it with details of a third-party JDBC driver class. This implementation has the following features: • Caches credentials for opening connections. • Supports multi-threading. • Compatible with the Spring transaction API. • Compatible with OSGi.
DriverManagerDataSource	*(Deprecated)* Incompatible with OSGi containers. This class is superseded by the **SimpleDriverDataSource**.

Data Source Class	Description
SingleConnectionDataSource	A data source that opens only one database connection (that is, every call to **getConnection()** returns a reference to the same connection instance). It follows that this data source is *incompatible with multi-threading* and is therefore not recommended for general use.

J2EE data source adapters

If your application is deployed into a J2EE container, it does not make sense to create a data source directly. Instead, you should let the J2EE container take care of creating data sources and you can then access those data sources by doing a JNDI lookup. For example, the following code fragment shows how you can obtain a data source from the JNDI reference, **java:comp/env/jdbc/myds**, and then wrap the data source with a **UserCredentialsDataSourceAdapter**.

```
<bean id="myTargetDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="java:comp/env/jdbc/myds"/>
</bean>

<bean id="myDataSource"
class="org.springframework.jdbc.datasource.UserCredentialsDataSourceAdapter">
  <property name="targetDataSource" ref="myTargetDataSource"/>
  <property name="username" value="myusername"/>
  <property name="password" value="mypassword"/>
</bean>
```

The **JndiObjectFactoryBean** exploits the Spring bean factory pattern to look up an object in JNDI. When this bean's ID, **myTargetDataSource**, is referenced elsewhere in Spring using the **ref** attribute, instead of getting a reference to the **JndiObjectFactoryBean** bean, you actually get a reference to the bean that was looked up in JNDI (a **javax.sql.DataSource** instance).

The standard **javax.sql.DataSource** interface exposes two methods for creating connections: **getConnection()** and **getConnection(String username, String password)**. If (as is normally the case) the referenced database requires credentials in order to open a connection, the **UserCredentialsDataSourceAdapter** class provides a convenient way of ensuring that these user credentials are available. You can use this adapter class for wrapping JNDI-provided data sources that do not have their own credentials cache.

In addition to **UserCredentialsDataSourceAdapter**, there are a number of other adapter classes that you can use to wrap data sources obtained from JNDI lookups. These J2EE data source adapters are summarized in Table 2.4, “J2EE Data Source Adapters”.

Table 2.4. J2EE Data Source Adapters

Data Source Adapter	Description

Data Source Adapter	Description
UserCredentialsDataSourceAdapter	Data source wrapper class that caches username/password credentials, for cases where the wrapped data source does not have its own credentials cache. This class can be used to wrap a data source obtained by JNDI lookup (typically, in a J2EE container). The username/password credentials are bound to a specific thread. Hence, you can store different connection credentials for different threads.
IsolationLevelDataSourceAdapter	Subclass of **UserCredentialsDataSourceAdapter** which, in addition to caching user credentials, also applies the current Spring transaction's level of isolation to all of the connections it creates.
WebSphereDataSourceAdapter	Same functionality as **IsolationLevelDataSourceAdapter**, except that the implementation is customized to work with IBM-specific APIs.

Data source proxies for special features

You can wrap a data source with a *data source proxy* in order to add special functionality to a data source. The data source proxies can be applied either to a standalone data source or a data source provided by the container. They are summarized in Table 2.5, “Data Source Proxies”.

Table 2.5. Data Source Proxies

Data Source Proxy	Description
LazyConnectionDataSourceProxy	This proxy uses lazy semantics to avoid unnecessary database operations. That is, a connection will not actually be opened until the application code attempts to write (or read) to the database. For example, if some application code opens a connection, begins a transaction, and then commits a transaction, but never actually accesses the database, the lazy connection proxy would optimize these database operations away.
TransactionAwareDataSourceProxy	Provides support for legacy database code that is *not* implemented using the Spring persistence API. *Do not use this proxy for normal transaction support.* The other Spring data sources are already compatible with the Spring persistence and transaction APIs. For example, if your application code uses Spring's **JdbcTemplate** class to access JDBC resources, do *not* use this proxy class.

Third-party JDBC driver managers

Table 2.6, “Connection Details for Various Databases” shows the JDBC connection details for a variety of different database products.

Table 2.6. Connection Details for Various Databases

Database	JDBC Driver Manager Properties

HSQLDB	The JDBC driver class for HSQLDB is as follows:
	
	`org.hsqldb.jdbcDriver`
	
	To connect to a HSQLDB database, you can use one of the following JDBC URL formats:
	
	`jdbc:hsqldb:hsql[s]://host[:port][/DBName]`
	`[KeyValuePairs]`
	`jdbc:hsqldb:http[s]://host[:port][/DBName]`
	`[KeyValuePairs]`
	`jdbc:hsqldb:mem:DBName[KeyValuePairs]`
	
	Where the **hsqls** and **https** protocols use TLS security and the **mem** protocol references an in-process, transient database instance (useful for testing). For more details, see http://www.hsqldb.org/doc/src/.

MySQL	The JDBC driver class for MySQL is as follows:
	
	`com.mysql.jdbc.Driver`
	
	To connect to a MySQL database, use the following JDBC URL format:
	
	`jdbc:mysql://[host][,failoverhost...]`
	`[:port]/[DBName][Options]`
	
	Where the *Options* coincidentally have the same format as Camel component options—for example, ***?Option1=Value1&Option2=Value2***. For more details, see http://dev.mysql.com/doc/connector-j/en/connector-j-reference-configuration-properties.html.

Database	JDBC Driver Manager Properties
Oracle	Depending on which version of Oracle you are using choose one of the following JDBC driver classes: oracle.jdbc.OracleDriver *(Oracle 9i, 10)* oracle.jdbc.driver.OracleDriver *(Oracle 8i)* To connect to an Oracle database, use the following JDBC URL format: jdbc:oracle:thin:[user/password]@[host][:port]:*SID* Where the Oracle System ID (*SID*) identifies an Oracle database instance. For more details, see http://download.oracle.com/docs/cd/B10501_01/java.920/a96654/basic.htm.
DB2	The JDBC driver class for DB2 is as follows: com.ibm.db2.jcc.DB2Driver To connect to a DB2 database, use the following JDBC URL format: jdbc:db2://host[:port]/*DBName*
SQL Server	The JDBC driver class for SQL Server is as follows: com.microsoft.jdbc.sqlserver.SQLServerDriver To connect to a SQL Server database, use the following JDBC URL format: jdbc:microsoft:sqlserver://host[:port];DatabaseName=*DBName*
Sybase	The JDBC driver class for Sybase is as follows: com.sybase.jdbc3.jdbc.SybDriver To connect to a Sybase database, use the following JDBC URL format: jdbc:sybase:Tds:host:port/*DBName*

Database	JDBC Driver Manager Properties
Informix	The JDBC driver class for Informix is as follows: com.informix.jdbc.IfxDriver To connect to an Informix database, use the following JDBC URL format: jdbc:informix-sqli://host:port/*DBName*:informixserver=*DBServerName*
PostgreSQL	The JDBC driver class for PostgreSQL is as follows: org.postgresql.Driver To connect to a PostgreSQL database, use the following JDBC URL format: jdbc:postgresql://host[:port]/*DBName*
MaxDB	The JDBC driver class for the SAP database is as follows: com.sap.dbtech.jdbc.DriverSapDB To connect to a MaxDB database, use the following JDBC URL format: jdbc:sapdb://host[:port]/*DBName*
FrontBase	The JDBC driver class for FrontBase is as follows: com.frontbase.jdbc.FBJDriver To connect to a FrontBase database, use the following JDBC URL format: jdbc:FrontBase://host[:port]/*DBName*

2.6.2. Hibernate

Overview

To enable transactions while accessing Hibernate objects, you need to provide an instance of the Hibernate transaction manager, of **HibernateTransactionManager** type, as described here. You can then use the **transacted()** DSL command to create a transaction scope in a route.

Sample Hibernate configuration

Example 2.3, “Hibernate Transaction Manager Configuration” shows how to instantiate a Hibernate transaction manager, of **HibernateTransactionManager** type, which is required if you want to integrate Hibernate object-oriented persistence with Spring transactions. The Hibernate transaction manager requires a reference to a Hibernate session factory, and the Hibernate session factory takes a reference to a JDBC data source.

Example 2.3. Hibernate Transaction Manager Configuration

```
<beans ... >
    ...
    <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
        <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
        <property name="username" value="sa"/>
        <property name="password" value=""/>
    </bean>

    <bean id="mySessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="myDataSource"/>
        <property name="mappingResources">
            <list>
                <value>product.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.dialect=org.hibernate.dialect.HSQLDialect
            </value>
        </property>
    </bean>

    <bean id="hibernateTxManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>
</beans>
```

Hibernate transaction manager bean

In Example 2.3, “Hibernate Transaction Manager Configuration”, the **hibernateTxManager** bean is a local Hibernate transaction manager instance, of **HibernateTransactionManager** type. There is just one property you need to provide to the Hibernate transaction manager: a reference to a *Hibernate session factory*.

Hibernate session factory bean

In Example 2.3, “Hibernate Transaction Manager Configuration”, the **mySessionFactory** bean is a Hibernate session factory of **org.springframework.orm.hibernate3.LocalSessionFactory** type. This session factory bean is needed by the Hibernate transaction manager.

In general, you need to supply the following properties to a Hibernate **LocalSessionFactory** bean instance:

dataSource

An instance of **javax.sql.DataSource**, which is the JDBC data source of the database that Hibernate is layered over. For details of how to configure a JDBC data source, see Section 2.6.1, “JDBC Data Source”.

mappingResources

Specifies a list of one or more *mapping association* files on the class path. A Hibernate mapping association defines how Java objects map to database tables.

hibernateProperties

Allows you to set any Hibernate property, by supplying a list of property settings. The most commonly needed property is **hibernate.dialect**, which indicates to Hibernate what sort of database it is layered over, enabling Hibernate to optimize its interaction with the underlying database. The dialect is specified as a class name, which can have one of the following values:

org.hibernate.dialect.Cache71Dialect
org.hibernate.dialect.DataDirectOracle9Dialect
org.hibernate.dialect.DB2390Dialect
org.hibernate.dialect.DB2400Dialect
org.hibernate.dialect.DB2Dialect
org.hibernate.dialect.DerbyDialect
org.hibernate.dialect.FirebirdDialect
org.hibernate.dialect.FrontBaseDialect
org.hibernate.dialect.H2Dialect
org.hibernate.dialect.HSQLDialect
org.hibernate.dialect.IngresDialect
org.hibernate.dialect.InterbaseDialect
org.hibernate.dialect.JDataStoreDialect
org.hibernate.dialect.MckoiDialect
org.hibernate.dialect.MimerSQLDialect
org.hibernate.dialect.MySQL5Dialect
org.hibernate.dialect.MySQL5InnoDBDialect
org.hibernate.dialect.MySQLDialect
org.hibernate.dialect.MySQLInnoDBDialect
org.hibernate.dialect.MySQLMyISAMDialect
org.hibernate.dialect.Oracle9Dialect
org.hibernate.dialect.OracleDialect
org.hibernate.dialect.PointbaseDialect
org.hibernate.dialect.PostgreSQLDialect
org.hibernate.dialect.ProgressDialect
org.hibernate.dialect.RDMSOS2200Dialect
org.hibernate.dialect.SAPDBDialect
org.hibernate.dialect.SQLServerDialect
org.hibernate.dialect.Sybase11Dialect
org.hibernate.dialect.SybaseAnywhereDialect
org.hibernate.dialect.SybaseDialect
org.hibernate.dialect.TimesTenDialect

2.6.3. JPA

Overview

To enable transactions in a JPA component, you need to provide the JPA component with a reference to a transaction manager, of **JpaTransactionManager** type. The Java Persistence API is a generic wrapper API for object-relational persistence and it can be layered over a variety of different object-relational mapping technologies.

Sample JPA configuration

Example 2.4, “JPA Transaction Manager Configuration” shows how to customize the configuration of a JPA component (creating a component with the bean ID, jpa), so that the JPA component supports Spring transactions. When used with transactions, the JPA component requires a reference to an entity manager factory and a reference to a transaction manager.

Example 2.4. JPA Transaction Manager Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

   <bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent">
     <property name="entityManagerFactory" ref="entityManagerFactory"/>
     <property name="transactionManager" ref="jpaTxManager"/>
   </bean>

   <bean id="jpaTxManager" class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="entityManagerFactory"/>
   </bean>

   <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
      <property name="persistenceUnitName" value="camel"/>
   </bean>

</beans>
```

JPA transaction manager bean

In Example 2.4, “JPA Transaction Manager Configuration”, the **jpaTxManager** bean is a local JPA transaction manager instance, of **JpaTransactionManager** type. The JPA transaction manager requires a reference to an entity manager factory bean (in this example, the **entityManagerFactory** bean).

If you deploy your application into an OSGi container, however, you might want to consider using a **JtaTransactionManager** instead. See Table 2.2, “Global Transaction Managers”.

Entity manager factory bean

The entity manager factory bean encapsulates the JPA runtime functionality. For example, the Spring **LocalEntityManagerFactoryBean** class is just a wrapper around the standard **javax.persistence.EntityManagerFactory** class. The entity manager factory is used to create a

javax.persistence.EntityManager instance, where the entity manager is associated with a unique *persistence context*. A persistence context represents a consistent set of entity objects that are instantiated from the underlying database (analogous to a Hibernate session).

The **LocalEntityManagerFactoryBean** class is a relatively simple JPA wrapper class that is suitable for simple demonstrations and testing purposes. This class reads its required configuration information from the **persistence.xml** file, which is found at the standard location, **META-INF/persistence.xml**, on the class path (see the section called “Sample persistence.xml file”). The **persistenceUnitName** property references a section of the **persistence.xml** file.

JPA entity manager factories

As well as instantiating a **LocalEntityManagerFactoryBean** bean, there are other ways of obtaining a JPA entity manager factory, as summarized in Table 2.7, “Obtaining JPA Entity Manager Factory”.

Table 2.7. Obtaining JPA Entity Manager Factory

Entity Manager Factory	Description
Obtain from JNDI	If your application is deployed in a J2EE container, the recommended approach is to let the container take care of instantiating the entity manager factory. You can then obtain a reference to the entity manager factory using JNDI. See Obtaining an EntityManagerFactory from JNDI in the Spring documentation.
LocalEntityManagerFactoryBean	For simple standalone applications and for testing, the simplest option is to create a bean of this type. The JPA runtime is configured using the standard **META-INF/persistence.xml** file.
LocalContainerEntityManagerFactoryBean	Use this class, if you need to configure special bootstrap options for the JPA runtime. In spite of the name, this class is *not* restricted to containers; you can also use it in standalone mode. See LocalContainerEntityManagerFactoryBean in the Spring documentation.

JPA bootstrap contract

The JPA is a thin abstraction layer that allows you to write code that is independent of a particular object-relational mapping product—for example, it enables you to layer your application over products such as OpenJPA, Hibernate, or TopLink. To match the application code to a specific JPA implementation, JPA defines a *bootstrap contract*, which is a procedure to locate and configure JPA implementations, as follows:

- To make a JPA implementation available to your application, put the JAR file containing the relevant JPA provider class (of **javax.persistence.spi.PersistenceProvider** type) on your class path. In fact, it is possible to add multiple JPA providers to your class path: you can optionally specify which JPA provider to use in the **persistence.xml** file.
- The JPA persistence layer is configured by the standard **persistence.xml** file, which is normally located in **META-INF/persistence.xml** on the class path.

Sample persistence.xml file

Example 2.5, “Sample persistence.xml File” shows a sample **persistence.xml** file for configuring an OpenJPA JPA provider layered over a Derby database.

Example 2.5. Sample persistence.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             version="1.0"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

  <persistence-unit name="camel" transaction-type="RESOURCE_LOCAL">
    <!--
       The default provider can be OpenJPA, or some other product.
       This element is optional if OpenJPA is the only JPA provider
       in the current classloading environment, but can be specified
       in cases where there are multiple JPA implementations available.
    -->
(1)   <provider>
        org.apache.openjpa.persistence.PersistenceProviderImpl
    </provider>

(2)   <class>org.apache.camel.examples.MultiSteps</class>
      <class>org.apache.camel.examples.SendEmail</class>

(3)   <properties>
        <property name="openjpa.ConnectionURL" value="jdbc:derby:target/derby;create=true"/>
      <property name="openjpa.ConnectionDriverName"
value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema"/>
      <property name="openjpa.Log" value="DefaultLevel=WARN, Tool=INFO, SQL=TRACE"/>
    </properties>
  </persistence-unit>

</persistence>
```

(1) The **provider** element can be used to specify the OpenJPA provider implementation class. If the **provider** element is omitted, the JPA layer simply uses the first JPA provider it can find. Hence, it is recommended to specify the **provider** element, if there are multiple JPA providers on your class path.

To make a JPA provider available to an application, simply add the provider's JAR file to the class path and the JPA layer will auto-detect the JPA provider.

(2) Use the **class** elements to list all of the Java types that you want to persist using the JPA framework.

(3) Use the **properties** element to configure the underlying JPA provider. In particular, you should at least provide enough information here to configure the connection to the underlying database.

Sample annotated class

The following code example shows how the **org.apache.camel.examples.SendEmail** class referenced in Example 2.5, “Sample persistence.xml File” should be annotated to turn it into a persistent entity bean (so that it is persistible by JPA):

```
// Java
package org.apache.camel.examples;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

/**
 * Represents a task which is added to the database, then removed from the database when it is
consumed
 *
 * @version $Revision$
 */
@Entity
public class SendEmail {
    private Long id;
    private String address;

    public SendEmail() {
    }

    public SendEmail(String address) {
        setAddress(address);
    }

    @Override
    public String toString() {
        return "SendEmail[id: " + getId() + " address: " + getAddress() + "]";
    }

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

The preceding class has the following JPA annotations:

@javax.persistence.Entity

Specifies that the following class is persistible by the JPA.

@javax.persistence.Id

The following bean property must be used as the primary key (for locating objects of this type in the database).

@javax.persistence.GeneratedValue

Specifies that the primary key values should be automatically generated by the JPA runtime (you can optionally set attributes on this annotation to configure the ID generation algorithm as well).

For the complete list of JPA annotations, see the API for the **javax.persistence** package.

CHAPTER 3. JMS TRANSACTIONS

Abstract

JMS endpoints create special problems when transactions are enabled. Their behavior is effected by the type of transaction manager in use, the caching level in use, and the message exchange pattern in use.

NOTE

For tips on optimizing transaction performance, see Appendix A, *Optimizing Performance of JMS Single- and Multiple-Resource Transactions*.

3.1. CONFIGURING THE JMS COMPONENT

Overview

To enable transactions in a JMS component (thus enabling JMS endpoints to play the role either of a transactional resource or a transactional client), you need to:

- set the transacted property
- provide the JMS component with a reference to a suitable transaction manager

In addition, you may want to adjust the JMS component's cache level setting. External transaction managers can impact caching performance.

Camel JMS component configuration

The easiest way to configure a JMS endpoint to participate in transactions is to create a new an instance of a Camel JMS component that has the proper settings. To do so:

1. Create a **bean** element that has its **class** attribute set to **org.apache.camel.component.jms.JmsComponent**.

   This bean creates an instance of the JMS component.

2. Set the bean's **id** attribute to a unique, short, string.

   The id will be used to create route endpoint's that use this JMS component.

3. Add an empty **property** child to the bean.

4. Add a **name** attribute with the value of **configuration** to the **property** element.

5. Add a **ref** attribute whose value is the id of a **JmsConfiguration** bean to the **property** element.

   The **JmsConfiguration** bean is used to configure the JMS component.

6. Create a **bean** element that has its **class** attribute set to **org.apache.camel.component.jms.JmsConfiguration**.

   This bean creates an instance of the JMS component configuration.

7. Set the bean's **id** attribute to the value supplied for the **ref** attribute in Step 5.

8. Add a **property** child to the bean to configure the JMS connection factory.
   a. Set the **name** attribute to **connectionFactory**.
   b. Set the **ref** attribute to the id of a bean that configures a JMS connection factory.
9. Add an empty **property** child to the bean that specifies the transaction manager the component will use.
   a. Set the **name** attribute to **transactionManager**.
   b. Set the **ref** attribute to the id of a bean that configures transaction manager the endpoint will use.

   See Chapter 2, *Selecting a Transaction Manager*.
10. Add an empty **property** child to the bean that configures the component to participate in transactions.
    a. Set the **name** attribute to **transacted**.
    b. Set the **value** attribute to **true**.

       The transacted property determines if the endpoint can participate in transactions.
11. Optionally add an empty **property** child to the bean to change the default cache level.
    a. Set the **name** attribute to **cacheLevelName**.
    b. Set the **value** attribute to to a valid cache level. For details, see the section called “Cache levels and performance”.

The **JmsComponent** bean's **id** specifies the URI prefix used by JMS endpoints that will use the transactional JMS component. For example, in Example 3.1, “JMS Transaction Manager Configuration” the **JmsComponent** bean's **id** equals **jmstx**, so endpoint that use the configured JMS component use the **jmstx:** prefix.

The **JmsConfiguration** class supports a large number of other properties, which are essentially identical to the JMS URI options described in chapter "JMS" in "Apache Camel Component Reference" .

Cache levels and performance

The settings for JMS cache level can impact performance when you are using transactions. The default cache level is **CACHE_AUTO**. This default auto detects if an external transaction manager is in use and sets the cache level as follows:

- **CACHE_CONSUMER** if only local JMS resources are in use
- **CACHE_NONE** if an external transaction manager is in use

If your transaction manager does not require that caching be disabled, you can raise the cache level to improve performance. Consult your transaction manager's documentation to determine what caching level it can support. Then override the default cache level by setting the JMS component's cacheLevelName property to the new cache level.

See chapter "JMS" in "Apache Camel Component Reference" for information on setting the cache level of the JMS component.

Caching with ActiveMQ endpoints

When the transactional resource is ActiveMQ, you can use either **CACHE_CONNECTION** or **CACHE_CONSUMER** for local JMS transactions.

CACHE_CONSUMER improves performance significantly, but to avoid losing messages on failover:

- Do not set a transaction manager, and
- Set the lazyCreateTransactionManager property to **false** in your JMS configuration.

For an example route definition using **CACHE_CONNECTION**, see the section called “Example using CACHE_CONNECTION”. For an example route definition using **CACHE_CONSUMER**, see the section called “Example using CACHE_CONSUMER”.

IMPORTANT

Be careful when configuring caching with local transactions. The configurations shown in the section called “Example using CACHE_CONNECTION” and the section called “Example using CACHE_CONSUMER” are believed to be safe (that is, no risk of message loss) for configuring Camel routes where the consumer and producer endpoints both connect to the *same* broker instance. For other configurations involving local transactions, it is generally safer to disable caching (**CACHE_NONE**).

Example using CACHE_CONSUMER

This route definition routes messages between two destinations on the *same* broker using local transactions. Not setting a transaction manager makes Spring use only one JMS session and one transaction. The Camel route is fully transacted because the Camel consumer and producer endpoints are transacted.

```
<camel:camelContext trace="false" id="My.impl.CamelContext">
   <camel:route id="Queue2Backend">
      <camel:from uri="activemq:queue:TEST.amq.failover.inputQueue" />
      <camel:log message="Sending message to backend..."/>
      <camel:to uri="activemq:queue:TEST.amq.failover.backendQueue" />
   </camel:route>
</camel:camelContext>

<bean id="jmsConectionFactory" class="org.apache.activemq.pool.PooledConnectionFactory"
destroy-method="stop">
   <property name="connectionFactory">
      <bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
         <property name="brokerURL" value="failover:(tcp://localhost:61616?wireFormat.
               tightEncodingEnabled=false)?
initialReconnectDelay=10000&useExponentialBackOff=false
               &maxReconnectDelay=10000"/>
         <property name="userName" value="admin"/>
         <property name="password" value="admin"/>
      </bean>
   </property>
   <property name="maxConnections" value="1"/>
</bean>

<bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
   <property name="connectionFactory" ref="jmsConectionFactory"/>
```

```
    <property name="lazyCreateTransactionManager" value="false"/>
    <property name="transacted" value="true"/>
    <property name="concurrentConsumers" value="1"/>
    <property name="cacheLevelName" value="CACHE_CONSUMER"/>
</bean>

<bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="configuration" ref="jmsConfig"/>
</bean>
```

Example using CACHE_CONNECTION

Example 3.1, “JMS Transaction Manager Configuration” shows the configuration of a JMS component, **jmstx** that supports Spring transactions. The JMS component is layered over an embedded instance of Apache ActiveMQ and the transaction manager is an instance of **JmsTransactionManager**.

Example 3.1. JMS Transaction Manager Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">
  ...
  <bean id="jmstx" class="org.apache.camel.component.jms.JmsComponent">
    <property name="configuration" ref="jmsConfig" />
  </bean>

  <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
      <property name="connectionFactory" ref="jmsConnectionFactory"/>
      <property name="transactionManager" ref="jmsTransactionManager"/>
      <property name="transacted" value="true"/>
      <property name="cacheLevelName" value="CACHE_CONNECTION"/>
  </bean>

  <bean id="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="vm://broker1?brokerConfig=xbean:tutorial/activemq.xml"/>
  </bean>

</beans>
```

To use this JMS component in a route you would use the URI prefix jmstx: as shown in Example 3.2, “URI for Using Transacted JMS Endpoint”.

Example 3.2. URI for Using Transacted JMS Endpoint

```
from("jmstx:queue:rawStockQuotes")
    .process(myFormatter)
    .to("jmstx:queue:formattedStockQuotes");
```

3.2. INONLY MESSAGE EXCHANGE PATTERN

Overview

The type of exchange created by a JMS consumer endpoint depends on the value of the **JMSReplyTo** header in the incoming message. If the **JMSReplyTo** header is absent from the incoming message, the consumer endpoint produces exchanges with the *InOnly* message exchange pattern (MEP). For example, consider the following route that receives a stream of stock quotes from the queue, **queue:rawStockQuotes**, reformats the incoming messages, and then forwards them to another queue, **queue:formattedStockQuotes**.

```
from("jmstx:queue:rawStockQuotes")
    .process(myFormatter)
    .to("jmstx:queue:formattedStockQuotes");
```

Routes that process *InOnly* exchanges can easily be combined with transactions. In the preceding example, the JMS queues are accessed through the transactional JMS instance, **jmstx** (see Section 3.1, “Configuring the JMS Component”). The transaction initiated by the consumer endpoint, **jmstx:queue:rawStockQuotes**, ensures that each incoming message is reliably transmitted to the producer endpoint, **jmstx:queue:formattedStockQuotes**.

Enforcing the InOnly message exchange pattern

Typically, a route designed to work for *InOnly* exchanges does *not* work properly for *InOut* exchanges. Unfortunately, this leaves the route at the mercy of the external JMS client: if the client should accidentally set a **JMSReplyTo** header, the JMS consumer endpoint will create an *InOut* exchange, which could lead to errors in a route that is designed for *InOnly* exchanges.

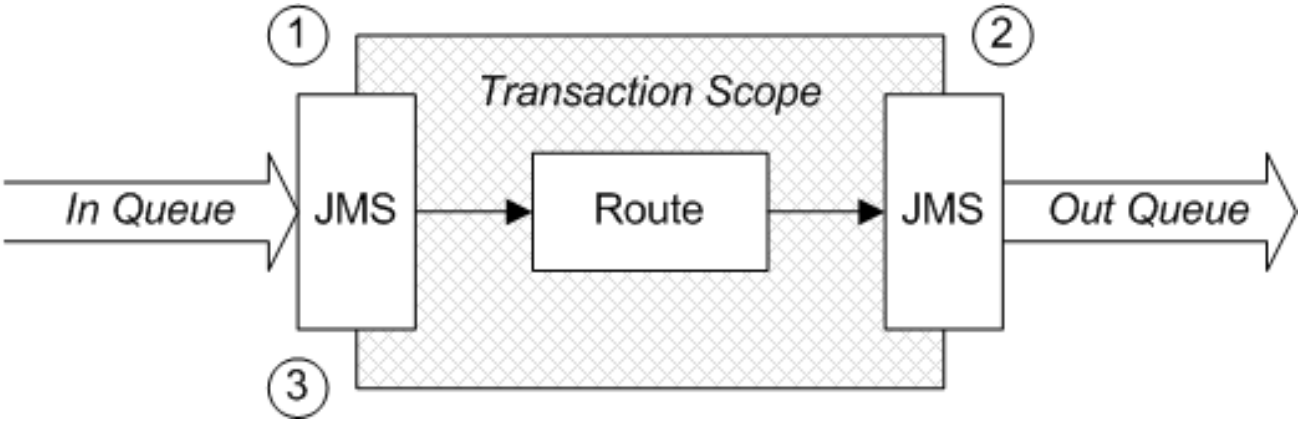
To avoid the risk of creating *InOut* exchanges when they are not wanted, you can use the **disableReplyTo** option in the JMS consumer to enforce the *InOnly* MEP. For example, the following route is guaranteed to process all incoming messages as *InOnly* exchanges:

```
from("jmstx:queue:rawStockQuotes?disableReplyTo=true")
    .process(myFormatter)
    .to("jmstx:queue:formattedStockQuotes");
```

InOnly scenario

Figure 3.1, “Transactional JMS Route that Processes InOnly Exchanges” shows an overview of a scenario consisting of JMS consumer endpoint feeding into a route that ends with a JMS producer endpoint. This route is designed to process exclusively *InOnly* exchanges.

Figure 3.1. Transactional JMS Route that Processes InOnly Exchanges



Description of InOnly scenario

Messages coming into the route shown in Figure 3.1, “Transactional JMS Route that Processes InOnly Exchanges” are processed as follows:

1. When a oneway message (**JMSReplyTo** header is absent) is polled by the JMS consumer endpoint, the endpoint starts a transaction, *provisionally* takes the message off the incoming queue, and creates an *InOnly* exchange object to hold the message.

2. After propagating through the route, the *InOnly* exchange arrives at the JMS producer endpoint, which *provisionally* writes the exchange to the outgoing queue.

3. At this point, we have arrived at the end of the transaction scope. If there were no errors (and the transaction is not marked for rollback), the transaction is automatically committed. Upon committing, both of the JMS endpoints send acknowledgement messages to the queues, turning the provisional read and the provisional write into a *committed* read and a *committed* write.

3.3. INOUT MESSAGE EXCHANGE PATTERN

Overview

Combining *InOut* mode with transactional JMS endpoints is problematic. In most cases, this mode of operation is fundamentally inconsistent and it is recommended that you refactor your routes to avoid this combination.

Enabling InOut mode in JMS

In a JMS consumer endpoint, *InOut* mode is automatically triggered by the presence of a **JMSReplyTo** header in an incoming JMS message. In this case, the endpoint creates an *InOut* exchange to hold the incoming message and it will use the **JMSReplyTo** queue to send the reply message.

Problems combining InOut mode with transactions

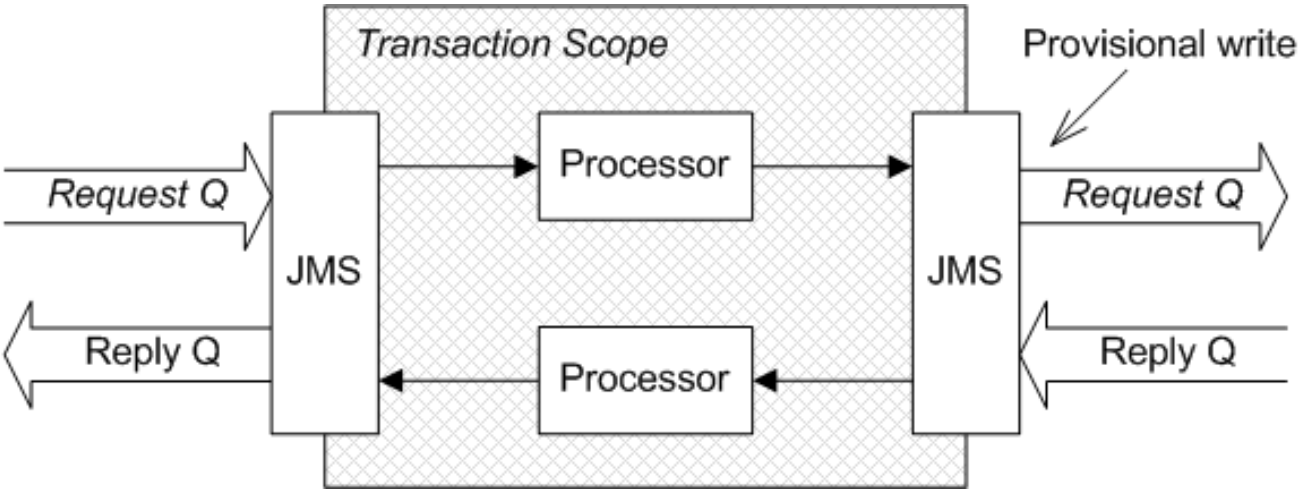
The *InOut* MEP is fundamentally incompatible with a route containing transactional JMS endpoints. In almost all cases, the route will hang and no reply will ever be sent. To understand why, consider the following route for processing payment requests:

```
from("jmstx:queue:rawPayments")
    .process(inputReformatter)
    .to("jmstx:queue:formattedPayments")
```

```
    .process(outputReformatter);
```

The JMS consumer endpoint, **jmstx:queue:rawPayments**, polls for messages, which are expected to have a **JMSReplyTo** header (for *InOut* mode). For each incoming message, a new transaction is started and an *InOut* exchange is created. After reformatting by the **inputReformatter** processor, the *InOut* exchange proceeds to the JMS producer endpoint, **jmstx:queue:formattedPayments**, which sends the message and expects to receive a reply on a temporary queue. This scenario is illustrated by Figure 3.2, "Transactional JMS Route that Processes InOut Exchanges"

Figure 3.2. Transactional JMS Route that Processes InOut Exchanges



The scope of the transaction includes the *entire* route, the request leg as well as the reply leg. The processing of the route proceeds as expected until the exchange arrives at the JMS producer endpoint, at which point the producer endpoint makes a *provisional* write to the outgoing request queue. At this point the route hangs: *the JMS producer endpoint is waiting to receive a message from the reply queue, but the reply can never be received because the outgoing request message was only* provisionally *written to the request queue* (and is thus invisible to the service at the other end of the queue).

It turns out that this problem is not trivial to solve. When you consider all of the ways that this scenario could fail and how to guarantee transactional integrity in all cases, it would require some substantial changes to the way that Apache Camel works. Fortunately, there is a simpler way of dealing with request/reply semantics that is already supported by Apache Camel.

Refactoring routes to avoid InOut mode

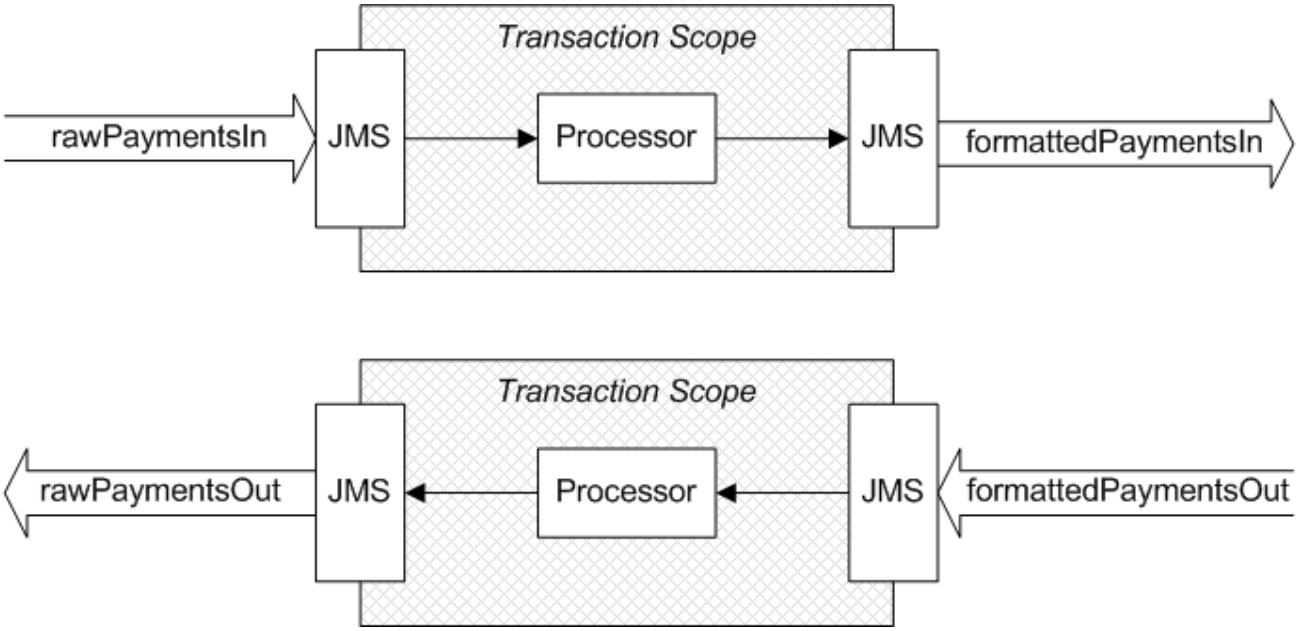
If you want to implement a transactional JMS route that has request/reply semantics, the easiest solution is to refactor your route to avoid using *InOut* exchanges. The basic idea is that instead of defining a single route that combines a request leg and a reply leg, you should refactor it into two routes: one for the (outbound) request leg and another for the (inbound) reply leg. For example, the payments example could be refactored into two separate routes as follows:

```
from("jmstx:queue:rawPaymentsIn")
    .process(inputReformatter)
    .to("jmstx:queue:formattedPaymentsIn");

from("jmstx:queue:formattedPaymentsOut")
    .process(outputReformatter)
    .to("jmstx:queue:rawPaymentsOut");
```

Instead of a single incoming queue, **queue:rawPayments**, which uses the queue from **JMSReplyTo** for replies, we now have a pair of queues: **queue:rawPaymentsIn**, for receiving incoming requests, and

queue:formattedPaymentsOut, for sending outgoing replies. Instead of a single outgoing queue, **queue:formattedPayments**, which implicitly uses a temporary queue for replies, we now have a pair of queues: **queue:formattedPaymentsOut**, for forwarding outgoing requests, and **queue:formattedPaymentsIn**, for receiving incoming replies. This scenario is illustrated by Figure 3.3, “Pair of Transactional JMS Routes that Support Request/Reply Semantics”.

Figure 3.3. Pair of Transactional JMS Routes that Support Request/Reply Semantics



A special case

There is a special case of a transactional JMS route where you *can* process *InOut* exchanges. If you look at the preceding examples, it is clear that the essential cause of deadlock in the route is the presence of JMS producer endpoints that obey request/reply semantics. In contrast to this, if you define a route where the JMS producer endpoints obey oneway semantics (fire-and-forget), deadlock does not occur.

For example, if you want to have a route that records all of the processed exchanges in a log queue, **queue:log**, you could define a route like the following:

```
from("jmstx:queue:inOutSource")
    .to(ExchangePattern.InOnly, "jmstx:queue:log")
    .process(myProcessor);
```

The exchanges coming into this route are of *InOut* type and both the consumer endpoint, **jmstx:queue:inOutSource**, and the producer endpoint, **jmstx:queue:log**, are transactional. The key to avoiding deadlock in this case is to force the producer endpoint to operate in oneway mode, by passing the **ExchangePattern.InOnly** parameter to the **to()** command,

CHAPTER 4. DATA ACCESS WITH SPRING

Abstract

If you are using transactions in your application, you will inevitably also be accessing some persistent resources. Spring provides a variety of APIs to support programmatic access to persistent resources and you might find it helpful to familiarize yourself with these data access APIs. In particular, this chapter describes Spring's JDBC API in some detail.

4.1. PROGRAMMING DATA ACCESS WITH SPRING TEMPLATES

Overview

To provide access to various kinds of persistent storage, Spring encapsulates the relevant API in a *template* class. The purpose of the template class is to provide a simplifying wrapper layer around each type of storage and to ensure that any required Spring features are integrated cleanly with the persistence layer.

Spring provides the following template classes for data access:

- JmsTemplate class.
- JdbcTemplate class.
- SimpleJdbcTemplate class.
- NamedParameterJdbcTemplate class.
- SqlMapClientTemplate class.
- HibernateTemplate class.
- JdoTemplate class.
- JpaTemplate class.

JmsTemplate class

The org.springframework.jms.core.JmsTemplate class is a general-purpose class for managing Java Messaging Service (JMS) connections. One of the main advantages of this class is that it simplifies the JMS synchronous access codes.

To create an instance of a **JmsTemplate**, you need to supply a reference to a javax.jms.ConnectionFactory object.

JdbcTemplate class

The org.springframework.jdbc.core.JdbcTemplate class is a wrapper around a JDBC data source, enabling you to access a JDBC database using SQL operations.

To create an instance of a **JdbcTemplate**, you need to supply a reference to a javax.sql.DataSource object (for example, see Section 2.6.1, “JDBC Data Source”).

NOTE

For a detailed discussion of the **JdbcTemplate** class, see Section 4.2, “Spring JDBC Template”.

SimpleJdbcTemplate class

The org.springframework.jdbc.core.simple.SimpleJdbcTemplate class is a convenience wrapper around the **JdbcTemplate** class. This class has been pared down so that it includes only the most commonly used template methods and it has been optimized to exploit Java 5 features.

NamedParameterJdbcTemplate class

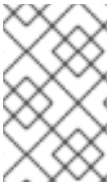
The org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate class is a convenience wrapper around the **JdbcTemplate** class, which enables you to use named parameters instead of the usual **?** placeholders embedded in a SQL statement.

SqlMapClientTemplate class

The org.springframework.orm.ibatis.SqlMapClientTemplate class is a simplifying wrapper around the iBATIS **SqlMapClient** class. iBATIS is an Object Relational Mapper (ORM) that is capable of automatically instantiating Java objects based on a given SQL database schema.

HibernateTemplate class

The org.springframework.orm.hibernate3.HibernateTemplate class provides an alternative to working with the raw Hibernate 3 session API (based on sessions returned from **SessionFactory.getCurrentSession()**).



NOTE

For Hibernate versions 3.0.1 or later, the Spring documentation recommends that you use the native Hibernate 3 API instead of the **HibernateTemplate** class, because transactional Hibernate access code can now be coded using the native Hibernate API.

JdoTemplate class

The org.springframework.orm.jdo.JdoTemplate class provides an alternative to working with the raw JDO PersistenceManager API. The main difference between the APIs relates to their exception handling. See the Spring JavaDoc for details.

JpaTemplate class

The org.springframework.orm.jpa.JpaTemplate class provides an alternative to working with the raw JPA **EntityManager** API..

NOTE

The Spring documentation now recommends that you use the native JPA programming interface instead of the **JpaTemplate** class. Considering that the JPA programming interface is itself a thin wrapper layer, there is little advantage to be had by adding another wrapper layer on top of it.

4.2. SPRING JDBC TEMPLATE

Overview

This section describes how to access a database through the Spring **JdbcTemplate** class and provides a code example that shows how to use the **JdbcTemplate** class in practice.

JdbcTemplate class

The org.springframework.jdbc.core.JdbcTemplate class is the key class for accessing databases through JDBC in Spring. It provides a complete API for executing SQL statements on the database at run time. The following kinds of SQL operations are supported by **JdbcTemplate**:

- Querying (**SELECT** operations).
- Updating (**INSERT**, **UPDATE**, and **DELETE** operations).
- Other SQL operations (all other SQL operations).

Querying

The JdbcTemplate query methods are used to send **SELECT** queries to the database. A variety of different query methods are supported, depending on how complicated the return values are.

The simplest case is where you expect the query to return a single value from a single row. In this case, you can use a type-specific query method to retrieve the single value. For example, if you want to retrieve the balance of a particular customer's account from the **accounts** table, you could use the following code:

```
// Java
int origAmount = jdbc.queryForInt(
    "select amount from accounts where name = ?",
    new Object[]{name}
);
```

The arguments to the SQL query are provided as a static array of objects, **Object[]{name}**. In this example, the **name** string is bound to the question mark, **?**, in the SQL query string. If there are multiple arguments to the query string (where each argument in the SQL string is represented by a question mark, **?**), you would provide an object array with multiple arguments—for example, **Object[]{arg1,arg2,arg3,...}**.

The next most complicated case is where you expect the query to return multiple values from a single row. In this case, you can use one of the **queryForMap()** methods to retrieve the contents of a single row. For example, to retrieve the complete account details from a single customer:

```
// Java
Map<String,Object> rowMap = jdbc.queryForMap(
    "select * from accounts where name = ?",
    new Object[]{name}
);
```

Where the returned map object, **rowMap**, contains one entry for each column, using the column name as the key.

The most general case is where you expect the query to return multiple values from multiple rows. In this case, you can use one of the **queryForList()** methods to return the contents of multiple rows. For example, to return all of the rows from the **accounts** table:

```
// Java
List<Map<String,Object> > rows = jdbc.queryForList(
    "select * from accounts"
);
```

In some cases, a more convenient way of returning the table rows is to provide a **RowMapper**, which automatically converts each row to a Java object. The return value of a query call would then be a list of Java objects. For example, the contents of the **accounts** table could be returned as follows:

```
// Java
List<Account> accountList = jdbc.query(
    "select * from accounts",
    new Object[]{},
    new RowMapper() {
        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            Account acc = new Account();
            acc.setName(rs.getString("name"));
            acc.setAmount(rs.getLong("amount"));
            return acc;
        }
    }
);
```

Where each **Account** object in the returned list encapsulates the contents of a single row.

Updating

The JdbcTemplate update methods are used to perform **INSERT**, **UPDATE**, or **DELETE** operations on the database. The update methods modify the database contents, but do not return any data from the database (apart from an integer return value, which counts the number of rows affected by the operation).

For example, the following update operation shows how to set the **amount** field in a customer's account:

```
// Java
jdbc.update(
    "update accounts set amount = ? where name = ?",
    new Object[] {newAmount, name}
);
```

Other SQL operations

For all other SQL operations, there is a general purpose **execute()** method. For example, you would use this method to execute a **create table** statement, as follows:

```
// Java
jdbc.execute("create table accounts (name varchar(50), amount int)");
```

Example application

To illustrate the database operations you can perform through the **JdbcTemplate** class, consider the *account service*, which provides access to bank account data stored in a database. It is assumed that the database is accessible through a JDBC data source and the account service is implemented by an **AccountService** class that exposes the following methods:

- **credit()**—add a specific amount of money to a named account.
- **debit()**—subtract a specific amount of money from a named account.

By combining credit and debit operations, it is possible to model money transfers, which can also be used to demonstrate key properties of transaction processing.

Format of money transfer orders

For the account service example, the money transfer orders have a simple XML format, as follows:

```
<transaction>
  <transfer>
    <sender>Major Clanger</sender>
    <receiver>Tiny Clanger</receiver>
    <amount>90</amount>
  </transfer>
</transaction>
```

When this money transfer order is executed, the amount of money specified in the **amount** element is debited from the **sender** account and credited to the **receiver** account.

CreateTable class

Before we can start performing any queries on the database, the first thing we need to do is to create an **accounts** table and populate it with some initial values. Example 4.1, “The CreateTable Class” shows the definition of the **CreateTable** class, which is responsible for intializing the **accounts** table.

Example 4.1. The CreateTable Class

```
// Java
package com.fusesource.demo.tx.jdbc.java;

import javax.sql.DataSource;

import org.apache.log4j.Logger;
import org.springframework.jdbc.core.JdbcTemplate;

public class CreateTable {
    private static Logger log = Logger.getLogger(CreateTable.class);

    protected DataSource dataSource;
    protected JdbcTemplate jdbc;

    public DataSource getDataSource() {
        return dataSource;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
```

```
    }

    public CreateTable(DataSource ds) {
        log.info("CreateTable constructor called");
        setDataSource(ds);
        setUpTable();
    }

    public void setUpTable() {
        log.info("About to set up table...");
        jdbc = new JdbcTemplate(dataSource);
        jdbc.execute("create table accounts (name varchar(50), amount int)");
        jdbc.update("insert into accounts (name,amount) values (?,?)",
                new Object[] {"Major Clanger", 2000}
        );
        jdbc.update("insert into accounts (name,amount) values (?,?)",
                new Object[] {"Tiny Clanger", 100}
        );
        log.info("Table created");
    }
}
```

Where the **accounts** table consists of two columns: **name**, a string value that records the account holder's name, and **amount**, a long integer that records the amount of money in the account. Because this example uses an ephemeral database, which exists only temporarily in memory, it is necessary to re-initialize the database every time the example runs. A convenient way to initialize the table is by instantiating a **CreateTable** bean in the Spring XML configuration, as follows:

```
<beans ...>
    <!-- datasource to the database -->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
        <property name="driverClass" value="org.hsqldb.jdbcDriver"/>
        <property name="url" value="jdbc:hsqldb:mem:camel"/>
        <property name="username" value="sa"/>
        <property name="password" value=""/>
    </bean>

    <!--  Bean to initialize table in the DB -->
    <bean id="createTable" class="com.fusesource.demo.tx.jdbc.java.CreateTable">
        <constructor-arg ref="dataSource" />
    </bean>
    ...
</beans>
```

As soon as the **createTable** bean is instantiated, the **accounts** table is ready for use. Note that a reference to the JDBC data source, **dataSource**, is passed to the **CreateTable()** constructor, because the data source is needed to create a **JdbcTemplate** instance.

AccountService class

Example 4.2, “The AccountService class” shows an outline of the **AccountService** class, *not* including the service methods that access the database. The class expects to receive a data source reference through dependency injection, which it then uses to create a **JdbcTemplate** instance.

Example 4.2. The AccountService class

```
package com.fusesource.demo.tx.jdbc.java;

import java.util.List;

import javax.sql.DataSource;


import org.apache.camel.Exchange;
import org.apache.camel.language.XPath;
import org.apache.log4j.Logger;
import org.springframework.jdbc.core.JdbcTemplate;


public class AccountService {
    private static Logger log = Logger.getLogger(AccountService.class);
    private JdbcTemplate jdbc;

    public AccountService() {
    }

    public void setDataSource(DataSource ds) {
        jdbc = new JdbcTemplate(ds);
    }
    ...
    // Service methods (see below)
    ...
}
```

You can conveniently instantiate an **AccountService** bean in Spring XML, using dependency injection to pass the data source reference, as follows:

```
<beans ...>
    <!-- Bean for account service -->
    <bean id="accountService" class="com.fusesource.demo.tx.jdbc.java.AccountService">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    ...
</beans>
```

AccountService.credit() method

The **credit()** method adds the specified amount of money, **amount**, to the specified account, **name** in the **accounts** database table, as follows:

```
    public void credit(
1           @XPath("/transaction/transfer/receiver/text()") String name,
            @XPath("/transaction/transfer/amount/text()") String amount
        )
    {
        log.info("credit() called with args name = " + name + " and amount = " + amount);
2       int origAmount = jdbc.queryForInt(
```

```
            "select amount from accounts where name = ?",
            new Object[]{name}
        );
        int newAmount = origAmount + Integer.parseInt(amount);

3       jdbc.update(
            "update accounts set amount = ? where name = ?",
            new Object[] {newAmount, name}
        );
    }
```

1 For methods invoked using the **beanRef()** (or **bean()**) DSL command, Apache Camel provides a powerful set of annotations for binding the exchange to the method parameters. In this example, the parameters are annotated using the **@XPath** annotation, so that the result of the XPath expression is injected into the corresponding parameter.

For example, the first XPath expression, **/transaction/transfer/receiver/text()**, selects the contents of the **receiver** XML element from the body of the exchange's *In* message and injects them into the **name** parameter. Likewise, the contents of the **amount** element are injected into the **amount** parameter.

2 The **JdbcTemplate.queryForInt()** method returns the current balance of the **name** account. For details about using **JdbcTemplate** to make database queries, see the section called “Querying”.

3 The **JdbcTemplate.update()** method updates the balance of the **name** account, adding the specified amount of money. For details about using **JdbcTemplate** to make database updates, see the section called “Updating”.

AccountService.debit() method

The **debit()** method subtracts the specified amount of money, **amount**, from the specified account, **name** in the **accounts** database table, as follows:

```
    public void debit(
1           @XPath("/transaction/transfer/sender/text()") String name,
            @XPath("/transaction/transfer/amount/text()") String amount
        )
    {
        log.info("debit() called with args name = " + name + " and amount = " + amount);
        int iamount = Integer.parseInt(amount);
2       if (iamount > 100) {
            throw new IllegalArgumentException("Debit limit is 100");
        }
        int origAmount = jdbc.queryForInt(
            "select amount from accounts where name = ?",
            new Object[]{name}
        );
        int newAmount = origAmount - Integer.parseInt(amount);
3       if (newAmount < 0) {
            throw new IllegalArgumentException("Not enough in account");
        }

        jdbc.update(
            "update accounts set amount = ? where name = ?",
```

```
            new Object[] {newAmount, name}
        );
    }
```

1. The parameters of the **debit()** method are also bound to the exchange using annotations. In this case, however, the **name** of the account is bound to the **sender** XML element in the *In* message.

2. There is a fixed debit limit of 100. Amounts greater than this will trigger an **IllegalArgument** exception. This feature is useful, if you want to trigger a rollback to test a transaction example.

3. If the balance of the account would go below zero after debiting, abort the transaction by calling the **IllegalArgumentException** exception.

AccountService.dumpTable() method

The **dumpTable()** method is convenient for testing. It simply returns the entire contents of the **accounts** table as a string. It is implemented as follows:

```
public void dumpTable(Exchange ex) {
    log.info("dump() called");
    List<?> dump = jdbc.queryForList("select * from accounts");
    ex.getIn().setBody(dump.toString());
}
```

CHAPTER 5. TRANSACTION DEMARCATION

Abstract

Transaction demarcation refers to the procedures for starting, committing, and rolling back transactions. This chapter describes the mechanisms that are available for controlling transaction demarcation, both by programming and by configuration.

5.1. DEMARCATION BY MARKING THE ROUTE

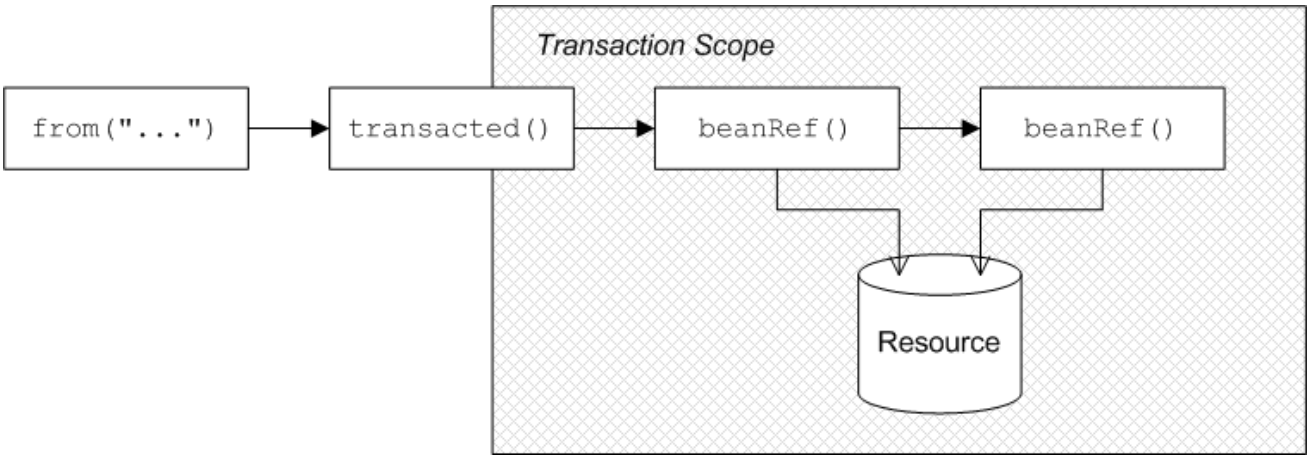
Overview

Apache Camel provides a simple mechanism for initiating a transaction in a route, by inserting the **transacted()** command in the Java DSL or by inserting the **<transacted/>** tag in the XML DSL.

Sample route with JDBC resource

Figure 5.1, “Demarcation by Marking the Route” shows an example of a route that is made transactional by adding the **transacted()** DSL command to the route. All of the route nodes following the**transacted()** node are included in the transaction scope. In this example, the two following nodes access a JDBC resource.

Figure 5.1. Demarcation by Marking the Route



The **transacted** processor demarcates transactions as follows: when an exchange enters the **transacted** processor, the **transacted** processor invokes the default transaction manager to *begin* a transaction (attaching it to the current thread); when the exchange reaches the end of the remaining route, the **transacted** processor invokes the transaction manager to *commit* the current transaction.

Route definition in Java DSL

The following Java DSL example shows how to define a transactional route by marking the route with the **transacted()** DSL command:

```
// Java
import org.apache.camel.spring.SpringRouteBuilder;

public class MyRouteBuilder extends SpringRouteBuilder {
    ...
```

```
    public void configure() {
        from("file:src/data?noop=true")
          .transacted()
          .beanRef("accountService","credit")
          .beanRef("accountService","debit");
    }
}
```

In this example, the **file** endpoint reads some files in XML format that describe a transfer of funds from one account to another. The first **beanRef()** invocation credits the specified sum of money to the beneficiary's account and then the second **beanRef()** invocation subtracts the specified sum of money from the sender's account. Both of the **beanRef()** invocations cause updates to be made to a database resource, which we are assuming is bound to the transaction through the transaction manager (for example, see Section 2.6.1, “JDBC Data Source”). For a sample implementation of the **accountService** bean, see Section 4.2, “Spring JDBC Template”.

Using SpringRouteBuilder

The **beanRef()** Java DSL command is available *only* in the **SpringRouteBuilder** class. It enables you to reference a bean by specifying the bean's Spring registry ID (for example, **accountService**). If you do not use the **beanRef()** command, you could inherit from the **org.apache.camel.builder.RouteBuilder** class instead.

Route definition in Spring XML

The preceding route can equivalently be expressed in Spring XML, where the **<transacted/>** tag is used to mark the route as transactional, as follows:

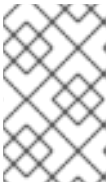
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ... >

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="file:src/data?noop=true"/>
      <transacted/>
      <bean ref="accountService" method="credit"/>
      <bean ref="accountService" method="debit"/>
    </route>
  </camelContext>

</beans>
```

Default transaction manager and transacted policy

To demarcate transactions, the **transacted** processor must be associated with a particular transaction manager instance. To save you having to specify the transaction manager every time you invoke **transacted()**, the **transacted** processor automatically picks a sensible default. For example, if there is only one instance of a transaction manager in your Spring configuration, the **transacted** processor implicitly picks this transaction manager and uses it to demarcate transactions.

A **transacted** processor can also be configured with a transacted policy, of **TransactedPolicy** type, which encapsulates a propagation policy and a transaction manager (see Section 5.3, “Propagation Policies” for details). The following rules are used to pick the default transaction manager or transaction policy:

1. If there is only one bean of **org.apache.camel.spi.TransactedPolicy** type, use this bean.



NOTE

The **TransactedPolicy** type is a base type of the **SpringTransactionPolicy** type that is described in Section 5.3, “Propagation Policies”. Hence, the bean referred to here could be a **SpringTransactionPolicy** bean.

2. If there is a bean of type, **org.apache.camel.spi.TransactedPolicy**, which has the ID, **PROPAGATION_REQUIRED**, use this bean.

3. If there is only one bean of **org.springframework.transaction.PlatformTransactionManager** type, use this bean.

You also have the option of specifying a bean explicitly by providing the bean ID as an argument to **transacted()**—see the section called “Sample route with PROPAGATION_NEVER policy in Java DSL”.

Transaction scope

If you insert a **transacted** processor into a route, a new transaction is created each time an exchange passes through this node and the transaction's scope is defined as follows:

1. The transaction is associated with the *current thread* only.

2. The transaction scope encompasses all of the route nodes *following* the **transacted** processor.

In particular, all of the route nodes *preceding* the **transacted** processor are not included in the transaction (but the situation is different, if the route begins with a transactional endpoint—see Section 5.2, “Demarcation by Transactional Endpoints”). For example, the following route is incorrect, because the **transacted()** DSL command mistakenly appears after the first **beanRef()** call (which accesses the database resource):

```
// Java
import org.apache.camel.spring.SpringRouteBuilder;

public class MyRouteBuilder extends SpringRouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
          .beanRef("accountService","credit")
          .transacted()  // <-- WARNING: Transaction started in the wrong place!
          .beanRef("accountService","debit");
    }
}
```

No thread pools in a transactional route

It is crucial to understand that a given transaction is associated with the *current thread* only. It follows that you *must not* create a thread pool in the middle of a transactional route, because the processing in the new threads will not participate in the current transaction. For example, the following route is bound to cause problems:

```
// Java
import org.apache.camel.spring.SpringRouteBuilder;
```

```
public class MyRouteBuilder extends SpringRouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
          .transacted()
          .threads(3)  // WARNING: Subthreads are not in transaction scope!
          .beanRef("accountService","credit")
          .beanRef("accountService","debit");
    }
}
```

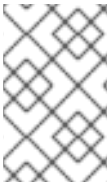
A route like the preceding one is certain to corrupt your database, because the **threads()** DSL command is incompatible with transacted routes. Even if the **threads()** call precedes the **transacted()** call, the route will not behave as expected.

Breaking a route into fragments

If you want to break a route into fragments and have each route fragment participate in the current transaction, you can use **direct:** endpoints. For example, to send exchanges to separate route fragments, depending on whether the transfer amount is big (greater than 100) or small (less than or equal to 100), you can use the **choice()** DSL command and **direct** endpoints, as follows:

```
// Java
import org.apache.camel.spring.SpringRouteBuilder;

public class MyRouteBuilder extends SpringRouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
            .transacted()
            .beanRef("accountService","credit")
            .choice().when(xpath("/transaction/transfer[amount > 100]"))
                .to("direct:txbig")
            .otherwise()
                .to("direct:txsmall");

        from("direct:txbig")
            .beanRef("accountService","debit")
            .beanRef("accountService","dumpTable")
            .to("file:target/messages/big");

        from("direct:txsmall")
            .beanRef("accountService","debit")
            .beanRef("accountService","dumpTable")
            .to("file:target/messages/small");
    }
}
```

Both the fragment beginning with **direct:txbig** and the fragment beginning with **direct:txsmall** participate in the current transaction, because the **direct** endpoints are *synchronous*. This means that the fragments execute in the same thread as the first route fragment and, therefore, they are included in the same transaction scope.



NOTE

You must not use **seda** endpoints to join the route fragments, because **seda** consumer endpoints create a new thread (or threads) to execute the route fragment (asynchronous processing). Hence, the fragments would not participate in the original transaction.

Resource endpoints

The following Apache Camel components act as *resource endpoints* when they appear as the destination of a route (for example, if they appear in the **to()** DSL command). That is, these endpoints can access a transactional resource, such as a database or a persistent queue. The resource endpoints can participate in the current transaction, as long as they are associated with the *same* transaction manager as the **transacted** processor that initiated the current transaction. If you need to access multiple resources, you must deploy your application in a J2EE container, which gives you access to a global transaction manager.

- JMS
- ActiveMQ
- AMQP
- JavaSpace
- JPA
- Hibernate
- iBatis
- JBI
- JCR
- JDBC
- LDAP

Sample route with resource endpoints

For example, the following route sends the order for a money transfer to two different JMS queues: the **credits** queue processes the order to credit the receiver's account; and the debits queue processes the order to **debit** the sender's account. Since there must only be a credit, if there is a corresponding debit, it makes sense to enclose the enqueueing operations in a single transaction. If the transaction succeeds, both the credit order and the debit order will be enqueued, but if an error occurs, *neither* order will be enqueued.

```
from("file:src/data?noop=true")
    .transacted()
    .to("jmstx:queue:credits")
    .to("jmstx:queue:debits");
```

5.2. DEMARCATION BY TRANSACTIONAL ENDPOINTS

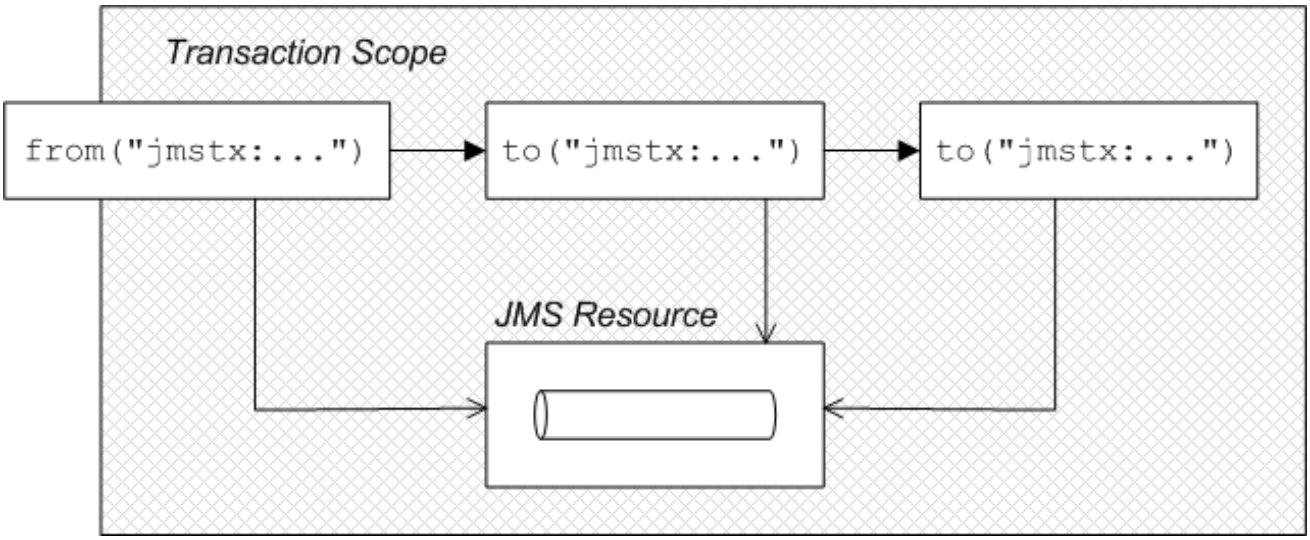
Overview

If a consumer endpoint at the start of a route accesses a resource, the **transacted()** command is of no use, because it initiates the transaction *after* an exchange is polled. In other words, the transaction starts too late to include the consumer endpoint within the transaction scope. The correct approach in this case is to make the endpoint itself responsible for initiating the transaction. An endpoint that is capable of managing transactions is known as a *transactional endpoint*.

Sample route with JMS endpoint

Figure 5.2, “Demarcation by Transactional Endpoints” shows an example of a route that is made transactional by the presence of a transactional endpoint at the start of the route (in the **from()** command). All of the route nodes are included in the transaction scope. In this example, all of the endpoints in the route access a JMS resource.

Figure 5.2. Demarcation by Transactional Endpoints



There are two different models of demarcation by transactional endpoint, as follows:

- *General case*—normally, a transactional endpoint demarcates transactions as follows: when an exchange arrives at the endpoint (or when the endpoint successfully polls for an exchange), the endpoint invokes its associated transaction manager to begin a transaction (attaching it to the current thread); and when the exchange reaches the end of the route, the transactional endpoint invokes the transaction manager to *commit* the current transaction.
- *JMS endpoint with InOut exchange*—when a JMS consumer endpoint receives an *InOut* exchange and this exchange is routed to another JMS endpoint, this must be treated as a special case. The problem is that the route can deadlock, if you try to enclose the entire request/reply exchange in a single transaction. For details of how to resolve this problem, see Section 3.3, “InOut Message Exchange Pattern”.

Route definition in Java DSL

The following Java DSL example shows how to define a transactional route by starting the route with a transactional endpoint:

```
from("jmstx:queue:giro")
    .to("jmstx:queue:credits")
    .to("jmstx:queue:debits");
```

Where the transaction scope encompasses the endpoints, **jmstx:queue:giro**, **jmstx:queue:credits**, and **jmstx:queue:debits**. If the transaction succeeds, the exchange is permanently removed from the **giro** queue and pushed on to the **credits** queue and the **debits** queue; if the transaction fails, the exchange does *not* get put on to the **credits** and **debits** queues and the exchange is pushed back on to the **giro** queue (by default, JMS will automatically attempt to redeliver the message).

The JMS component bean, **jmstx**, must be explicitly configured to use transactions, as follows:

```
<beans ...>
  <bean id="jmstx" class="org.apache.camel.component.jms.JmsComponent">
    <property name="configuration" ref="jmsConfig" />
  </bean>

  <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
      <property name="connectionFactory" ref="jmsConnectionFactory"/>
      <property name="transactionManager" ref="jmsTransactionManager"/>
      <property name="transacted" value="true"/>
  </bean>
  ...
</beans>
```

Where the transaction manager instance, **jmsTransactionManager**, is associated with the JMS component and the **transacted** property is set to **true** to enable transaction demarcation for *InOnly* exchanges. For the complete Spring XML configuration of this component, see Example 3.1, “JMS Transaction Manager Configuration”.

Route definition in Spring XML

The preceding route can equivalently be expressed in Spring XML, as follows:

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

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="jmstx:queue:giro"/>
      <to uri="jmstx:queue:credits"/>
      <to uri="jmstx:queue:debits"/>
    </route>
  </camelContext>

</beans>
```

transacted() not required

The **transacted()** DSL command is *not* required in a route that starts with a transactional endpoint. Nevertheless, assuming that the default transaction policy is **PROPAGATION_REQUIRED** (see Section 5.3, “Propagation Policies”), it is usually harmless to include the **transacted()** command, as in this example:

```
from("jmstx:queue:giro")
    .transacted()
    .to("jmstx:queue:credits")
    .to("jmstx:queue:debits");
```

However, it is possible for this route to behave in unexpected ways—for example, if a single **TransactedPolicy** bean having a non-default propagation policy is created in Spring XML (see the section called "Default transaction manager and transacted policy")—so it is generally better *not* to include the **transacted()** DSL command in routes that start with a transactional endpoint.

Transactional endpoints

The following Apache Camel components act as *transactional endpoints* when they appear at the start of a route (for example, if they appear in the **from()** DSL command). That is, these endpoints can be configured to behave as a transactional client and they can also access a transactional resource.

- JMS
- ActiveMQ
- AMQP
- JavaSpace
- JPA

5.3. PROPAGATION POLICIES

Overview

If you want to influence the way a transactional client creates new transactions, you can do so by specifying a *transaction policy* for it. In particular, Spring transaction policies enable you to specify a *propagation behavior* for your transaction. For example, if a transactional client is about to create a new transaction and it detects that a transaction is already associated with the current thread, should it go ahead and create a new transaction, suspending the old one? Or should it simply let the existing transaction take over? These kinds of behavior are regulated by specifying the propagation behavior on a transaction policy.

Transaction policies are instantiated as beans in Spring XML. You can then reference a transaction policy by providing its bean ID as an argument to the **transacted()** DSL command. For example, if you want to initiate transactions subject to the behavior, **PROPAGATION_REQUIRES_NEW**, you could use the following route:

```
from("file:src/data?noop=true")
    .transacted("PROPAGATION_REQUIRES_NEW")
    .beanRef("accountService","credit")
    .beanRef("accountService","debit")
    .to("file:target/messages");
```

Where the **PROPAGATION_REQUIRES_NEW** argument specifies the bean ID of a transaction policy bean that is configured with the **PROPAGATION_REQUIRES_NEW** behavior (see Example 5.1, "Transaction Policy Beans").

Spring transaction policies

Apache Camel lets you define Spring transaction policies using the **org.apache.camel.spring.spi.SpringTransactionPolicy** class (which is essentially a wrapper around a native Spring class). The **SpringTransactionPolicy** class encapsulates two pieces of data:

- A reference to a transaction manager (of **PlatformTransactionManager** type).
- A propagation behavior.

For example, you could instantiate a Spring transaction policy bean with **PROPAGATION_MANDATORY** behavior, as follows:

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

Propagation behaviors

The following propagation behaviors are supported by Spring (where these values were originally modelled on the propagation behaviors supported by J2EE):

PROPAGATION_MANDATORY

Support a current transaction; throw an exception if no current transaction exists.

PROPAGATION_NESTED

Execute within a nested transaction if a current transaction exists, else behave like **PROPAGATION_REQUIRED**.

NOTE

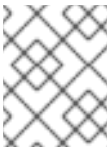
Nested transactions are not supported by all transaction managers.

PROPAGATION_NEVER

Do not support a current transaction; throw an exception if a current transaction exists.

PROPAGATION_NOT_SUPPORTED

Do not support a current transaction; rather always execute non-transactionally.



NOTE

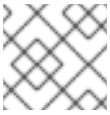
This policy requires the current transaction to be suspended, a feature which is not supported by all transaction managers.

PROPAGATION_REQUIRED

(Default) Support a current transaction; create a new one if none exists.

PROPAGATION_REQUIRES_NEW

Create a new transaction, suspending the current transaction if one exists.



NOTE

Suspending transactions is not supported by all transaction managers.

PROPAGATION_SUPPORTS

Support a current transaction; execute non-transactionally if none exists.

Defining policy beans in Spring XML

Example 5.1, “Transaction Policy Beans” shows how to define transaction policy beans for all of the supported propagation behaviors. For convenience, each of the bean IDs matches the specified value of the propagation behavior value, but in practice you can use whatever value you like for the bean IDs.

Example 5.1. Transaction Policy Beans

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">
  ...
  <bean id="PROPAGATION_MANDATORY
"class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager"/>
    <property name="propagationBehaviorName" value="PROPAGATION_MANDATORY"/>
  </bean>

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

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

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

  <!-- This is the default behavior. -->
  <bean id="PROPAGATION_REQUIRED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager"/>
  </bean>
```

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

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

</beans>
```

NOTE

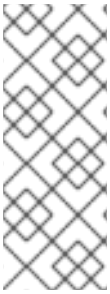
If you want to paste any of these bean definitions into your own Spring XML configuration, remember to customize the references to the transaction manager. That is, replace references to **txManager** with the actual ID of your transaction manager bean.

Sample route with PROPAGATION_NEVER policy in Java DSL

A simple way of demonstrating that transaction policies have some effect on a transaction is to insert a **PROPAGATION_NEVER** policy into the middle of an existing transaction, as shown in the following route:

```
from("file:src/data?noop=true")
    .transacted()
    .beanRef("accountService","credit")
    .transacted("PROPAGATION_NEVER")
    .beanRef("accountService","debit");
```

Used in this way, the **PROPAGATION_NEVER** policy inevitably aborts every transaction, leading to a transaction rollback. You should easily be able to see the effect of this on your application.



NOTE

Remember that the string value passed to **transacted()** is a bean ID, *not* a propagation behavior name. In this example, the bean ID is chosen to be the same as a propagation behavior name, but this need not always be the case. For example, if your application uses more than one transaction manager, you might end up with more than one policy bean having a particular propagation behavior. In this case, you could not simply name the beans after the propagation behavior.

Sample route with PROPAGATION_NEVER policy in Spring XML

The preceding route can be also be defined in Spring XML, as follows:

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

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

```
    <route>
      <from uri="file:src/data?noop=true"/>
      <transacted/>
      <bean ref="accountService" method="credit"/>
      <transacted ref="PROPAGATION_NEVER"/>
      <bean ref="accountService" method="debit"/>
    </route>
  </camelContext>

</beans>
```

5.4. ERROR HANDLING AND ROLLBACKS

Overview

While you can use standard Apache Camel error handling techniques in a transactional route, it is important to understand the interaction between exceptions and transaction demarcation. In particular, you need to bear in mind that thrown exceptions usually cause transaction rollback.

How to roll back a transaction

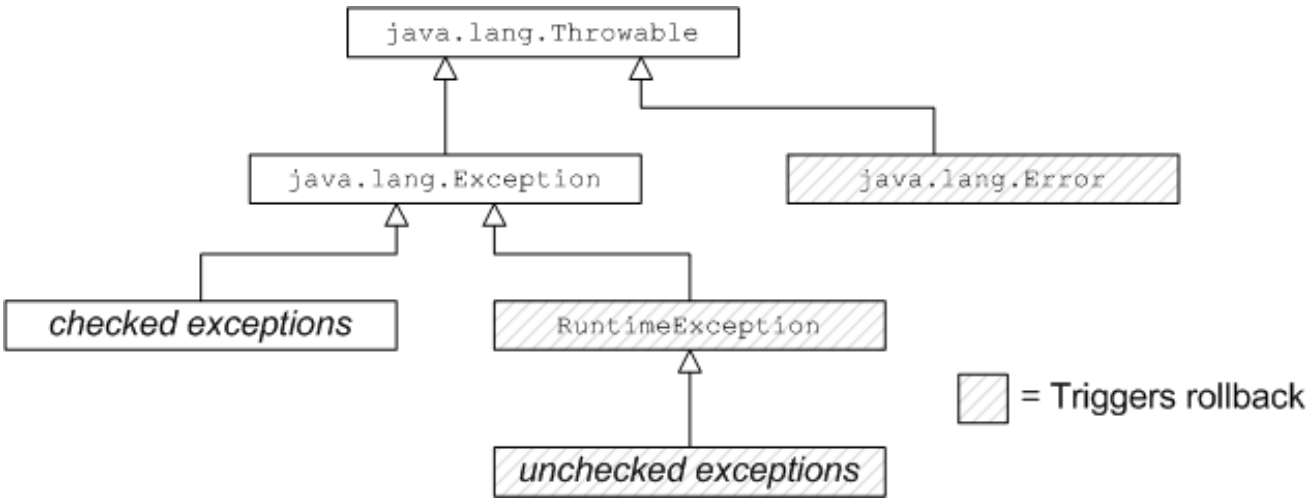
You can use one of the following approaches to roll back a transaction:

- the section called “Runtime exceptions as rollbacks”.
- the section called “The rollback() DSL command”.
- the section called “The markRollbackOnly() DSL command”.

Runtime exceptions as rollbacks

The most common way to roll back a Spring transaction is to *throw a runtime (unchecked) exception*—that is, where the exception is an instance or subclass of **java.lang.RuntimeException**. Java errors, of **java.lang.Error** type, also trigger transaction rollback. Checked exceptions, on the other hand, do *not* trigger rollback. Figure 5.3, “Errors and Exceptions that Trigger Rollback” summarises how Java errors and exceptions affect transactions, where the classes that trigger rollback are shaded gray.

Figure 5.3. Errors and Exceptions that Trigger Rollback



NOTE

The Spring framework also provides a system of XML annotations that enable you to specify which exceptions should or should not trigger rollbacks. For details, see Rolling back in the *Spring Reference Guide*.

WARNING

If a runtime exception is handled *within* the transaction (that is, before the exception has the chance to percolate up to the code that does the transaction demarcation), the transaction will *not* be rolled back. See the section called “How to define a dead letter queue” for details.

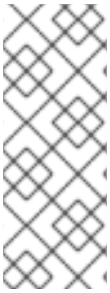
The rollback() DSL command

If you want to trigger a rollback in the middle of a transacted route, you can do this by calling the **rollback()** DSL command, which throws an **org.apache.camel.RollbackExchangeException** exception. In other words, the **rollback()** command uses the standard approach of throwing a runtime exception to trigger the rollback.

For example, if you decide that there should be an absolute limit on the size of money transfers in the account services application, you could trigger a rollback when the amount exceeds 100, using the following code:

Example 5.2. Rolling Back an Exception with rollback()

```
from("file:src/data?noop=true")
    .transacted()
    .beanRef("accountService","credit")
    .choice().when(xpath("/transaction/transfer[amount > 100]"))
        .rollback()
    .otherwise()
        .to("direct:txsmall");

from("direct:txsmall")
    .beanRef("accountService","debit")
    .beanRef("accountService","dumpTable")
    .to("file:target/messages/small");
```



NOTE

If you trigger a rollback in the preceding route, it will get trapped in an infinite loop. The reason for this is that the **RollbackExchangeException** exception thrown by **rollback()** propagates back to the file endpoint at the start of the route. The File component has a built-in reliability feature that causes it to resend any exchange for which an exception has been thrown. Upon resending, of course, the exchange just triggers another rollback, leading to an infinite loop.

The markRollbackOnly() DSL command

The **markRollbackOnly()** DSL command enables you to force the current transaction to roll back, *without* throwing an exception. This can be useful in cases where (as in Example 5.2, “Rolling Back an Exception with rollback()”) throwing an exception has unwanted side effects.

For example, Example 5.3, “Rolling Back an Exception with markRollbackOnly()” shows how to modify Example 5.2, “Rolling Back an Exception with rollback()” by replacing **rollback()** with **markRollbackOnly()**. This version of the route solves the problem of the infinite loop. In this case, when the amount of the money transfer exceeds 100, the current transaction is rolled back, but no exception is thrown. Because the file endpoint does not receive an exception, it does not retry the exchange, and the failed transactions is quietly discarded.

Example 5.3. Rolling Back an Exception with markRollbackOnly()

```
from("file:src/data?noop=true")
    .transacted()
    .beanRef("accountService","credit")
    .choice().when(xpath("/transaction/transfer[amount > 100]"))
        .markRollbackOnly()
    .otherwise()
        .to("direct:txsmall");
...
```

The preceding route implementation is not ideal, however. Although the route cleanly rolls back the transaction (leaving the database in a consistent state) and avoids the pitfall of infinite looping, it does *not* keep any record of the failed transaction. In a real-world application, you would typically want to keep track of any failed transaction. For example, you might want to write a letter to the relevant customer in order to explain why the transaction did not succeed. A convenient way of tracking failed transactions is to add a *dead-letter queue* to the route.

How to define a dead letter queue

In order to keep track of failed transactions, you can define an **onException()** clause, which enables you to divert the relevant exchange object to a dead-letter queue. When used in the context of transactions, however, you need to be careful about how you define the **onException()** clause, because of potential interactions between exception handling and transaction handling. Example 5.4, “How to Define a Dead Letter Queue” shows the correct way to define an **onException()** clause, assuming that you need to suppress the rethrown exception.

Example 5.4. How to Define a Dead Letter Queue

```
// Java
import org.apache.camel.spring.SpringRouteBuilder;

public class MyRouteBuilder extends SpringRouteBuilder {
    ...
    public void configure() {
        onException(IllegalArgumentException.class)
            .maximumRedeliveries(1)
            .handled(true)
            .to("file:target/messages?fileName=deadLetters.xml&fileExist=Append")
            .markRollbackOnly();  // NB: Must come *after* the dead letter endpoint.
```

```
        from("file:src/data?noop=true")
            .transacted()
            .beanRef("accountService","credit")
            .beanRef("accountService","debit")
            .beanRef("accountService","dumpTable")
            .to("file:target/messages");
    }
}
```

In the preceding example, **onException()** is configured to catch the **IllegalArgumentException** exception and send the offending exchange to a dead letter file, **deadLetters.xml** (of course, you can change this definition to catch whatever kind of exception arises in your application). The exception rethrow behavior and the transaction rollback behavior are controlled by the following special settings in the **onException()** clause:

- **handled(true)**—suppress the rethrown exception. In this particular example, the rethrown exception is undesirable because it triggers an infinite loop when it propagates back to the file endpoint (see the section called “The markRollbackOnly() DSL command”). In some cases, however, it might be acceptable to rethrow the exception (for example, if the endpoint at the start of the route does *not* implement a retry feature).
- **markRollbackOnly()**—marks the current transaction for rollback *without* throwing an exception. Note that it is essential to insert this DSL command *after* the **to()** command that routes the exchange to the dead letter queue. Otherwise, the exchange would never reach the dead letter queue, because **markRollbackOnly()** interrupts the chain of processing.

Catching exceptions around a transaction

Instead of using **onException()**, a simple approach to handling exceptions in a transactional route is to use the **doTry()** and **doCatch()** clauses around the route. For example, Example 5.5, “Catching Exceptions with doTry() and doCatch()” shows how you can catch and handle the **IllegalArgumentException** in a transactional route, without the risk of getting trapped in an infinite loop.

Example 5.5. Catching Exceptions with doTry() and doCatch()

```
// Java
import org.apache.camel.spring.SpringRouteBuilder;

public class MyRouteBuilder extends SpringRouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
            .doTry()
                .to("direct:split")
            .doCatch(IllegalArgumentException.class)
                .to("file:target/messages?fileName=deadLetters.xml&fileExist=Append")
            .end();

        from("direct:split")
            .transacted()
            .beanRef("accountService","credit")
            .beanRef("accountService","debit")
            .beanRef("accountService","dumpTable")
```

```
            .to("file:target/messages");
    }
}
```

In this example, the route is split into two segments. The first segment (from the **file:src/data** endpoint) receives the incoming exchanges and performs the exception handling using **doTry()** and **doCatch()**. The second segment (from the **direct:split** endpoint) does all of the transactional work. If an exception occurs within this transactional segment, it propagates first of all to the **transacted()** command, causing the current transaction to be rolled back, and it is then caught by the **doCatch()** clause in the first route segment. The **doCatch()** clause does *not* rethrow the exception, so the file endpoint does not do any retries and infinite looping is avoided.

CHAPTER 6. XA TRANSACTIONS IN RED HAT JBOSS FUSE

Abstract

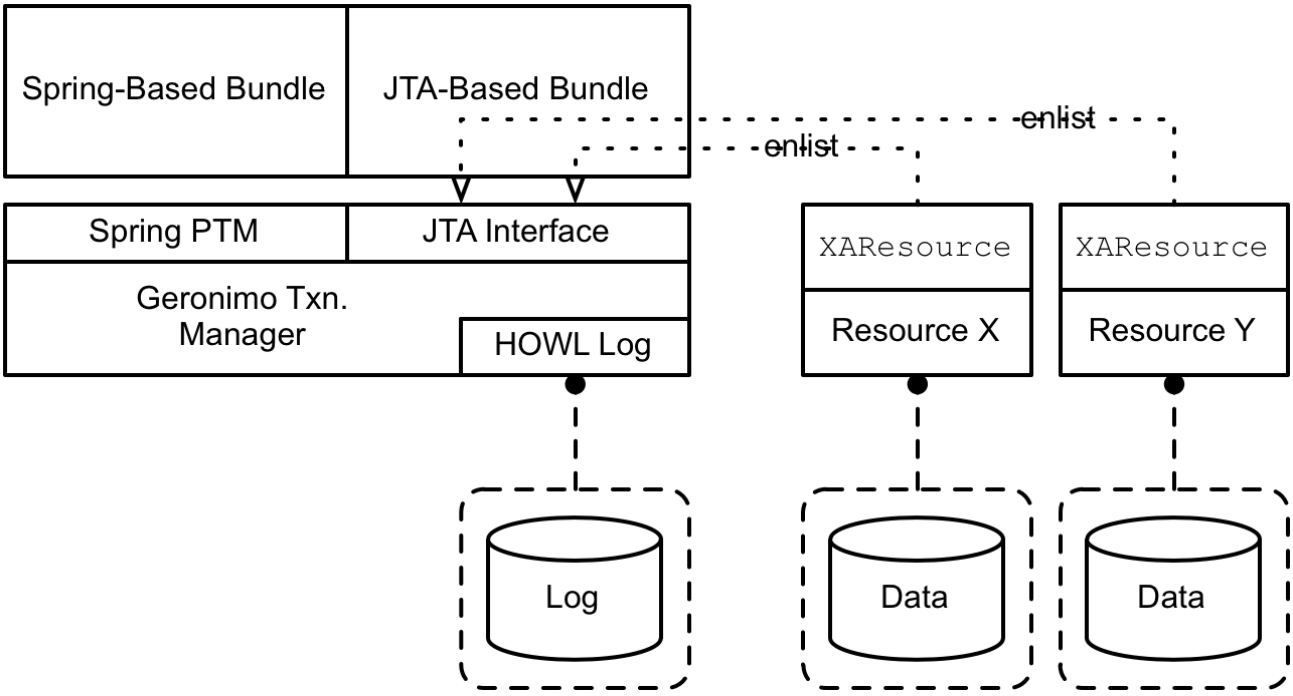
Red Hat JBoss Fuse provides a built-in global XA transaction manager, which applications can access if they need to coordinate transactions across multiple resources.

6.1. TRANSACTION ARCHITECTURE

Overview

Figure 6.1, “OSGi Transaction Architecture” shows an overview of the OSGi transaction architecture in Red Hat JBoss Fuse. The core of the architecture is a JTA transaction manager based on Apache Geronimo, which exposes various transaction interfaces as OSGi services.

Figure 6.1. OSGi Transaction Architecture



OSGi mandated transaction architecture

The *JTA Transaction Services Specification* section of the *OSGi Service Platform Enterprise Specification* describes the kind of transaction support that can (optionally) be provided by an OSGi container. Essentially, OSGi mandates that the transaction service is accessed through the Java Transaction API (JTA).

The transaction service exports the following JTA interfaces as OSGi services (the *JTA services*):

- **javax.transaction.UserTransaction**
- **javax.transaction.TransactionManager**
- **javax.transaction.TransactionSynchronizationRegistry**

Only one *JTA provider* should be made available in an OSGi container. In other words, the JTA services are registered only once and the objects obtained by importing references to the JTA services must be unique.

Spring transaction integration

The JBoss Fuse transaction service exports the following Spring interface as an OSGi service:

- **org.springframework.transaction.PlatformTransactionManager**

By obtaining a reference to the **PlatformTransactionManager** OSGi service, it is possible to integrate application bundles written using the Spring transaction API into the JBoss Fuse transaction architecture.

Red Hat JBoss Fuse transaction implementation

JBoss Fuse provides an OSGi-compliant implementation of the transaction service through the Apache Aries **transaction** feature, which consists mainly of the following bundles:

```
org.apache.aries.transaction.manager
org.apache.aries.transaction.wrappers
org.apache.aries.transaction.blueprint
```

The Aries transaction feature exports a variety of transaction interfaces as OSGi services (making them available to other bundles in the container), as follows:

- *JTA interfaces*—the JTA **UserTransaction**, **TransactionManager**, and **TransactionSynchronizationRegistry** interfaces are exported, as required by the OSGi transaction specification.
- *Spring transaction interface*—the Spring **PlatformTransactionManager** interface is exported, in order to facilitate bundles that are written using the Spring transaction APIs.

The **PlatformTransactionManager** OSGi service and the JTA services access the *same* underlying transaction manager.

Installing the transaction feature

To access the JBoss Fuse transaction implementation, you must install the transaction feature. In a standalone container, enter the following console command:

```
JBossFuse:karaf@root> features:install transaction
```

If you are deploying into a fabric, add the **transaction** feature to your application's profile.

Geronimo transaction manager

The underlying implementation of the JBoss Fuse transaction service is provided by the Apache Geronimo transaction manager. Apache Geronimo is a full implementation of a J2EE server and, as part of the J2EE implementation, Geronimo has developed a sophisticated transaction manager with the following features:

- Support for enlisting multiple XA resources.
- Support for 1-phase and 2-phase commit protocols.

- Support for suspending and resuming transactions.
- Support for automatic transaction recovery upon startup.

The transaction recovery feature depends on a *transaction log*, which records the status of all pending transactions in persistent storage.

Accessing Geronimo directly

Normally, the Geronimo transaction manager is accessed indirectly—for example, through the JTA wrapper layer or through the Spring wrapper layer. But if you need to access Geronimo directly, the following interface is also exposed as an OSGi service:

- **org.apache.geronimo.transaction.manager.RecoverableTransactionManager**

HOWL transaction log

The implementation of the transaction log is provided by HOWL, which is a high speed persistent logger that is optimized for XA transaction logs.

JTA-based application bundles

Normally, it is recommended that application bundles access the transaction service through the JTA interface. To use the transaction service in a JTA-based application bundle, import the relevant JTA service as an OSGi service and use the JTA service to begin, commit, or rollback transactions.

Spring-based application bundles

If you have already implemented an application bundle using the Spring transaction API, you might find it more convenient to access the transaction service through the Spring API (represented by the **PlatformTransactionManager** Java interface). This means that you are able to deploy the same source code either in a pure Spring container or in an OSGi container, by changing only the configuration snippet that obtains a reference to the transaction service.

References

The following specifications are relevant to the transaction architecture in JBoss Fuse:

- *OSGi transaction specification*—in section *123 JTA Transaction Services Specification v1.0* from the OSGi Service Platform Enterprise Specification v4.2
- Java Transaction API specification.
- *Spring transactions API*—see the Transaction Management chapter from the current Spring Reference Manual.

6.2. CONFIGURING THE TRANSACTION MANAGER

Overview

You can configure some basic parameters of the Aries transaction manager by editing the properties in the Aries transaction manager configuration file. In particular you must *always* specify the location of the HOWL log file directory and it is necessary to set the **aries.transaction.recoverable** property explicitly to **true**, if you want to enable the transaction recovery mechanism.

Configuration file

To configure the Aries transaction manager, you can edit the properties in the following configuration file:

EsbInstallDir/etc/org.apache.aries.transaction.cfg

Transaction manager properties

The properties that you can set in the **org.apache.aries.transaction.cfg** file include the following:

aries.transaction.recoverable

A boolean variable that specifies whether or not the transaction manager is recoverable. If not set, it defaults to **false**.

aries.transaction.timeout

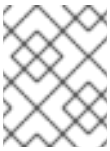
Specifies the transaction timeout in seconds. Default is 600 (that is, 10 minutes).

aries.transaction.tmid

Specifies the transaction manager identification string that gets appended to all transaction XIDs. This identification string allows transactions from different transaction managers to be disambiguated during transaction recovery, and should be different for each JBoss Fuse container that performs global transactions on a particular set of transactional resources. The string can be up to 64 characters in length. If not specified, a default identification string would be used, but this default value is the same for all JBoss Fuse containers.

aries.transaction.howl.bufferSize

Specifies the HOWL log buffer size in units of KiB, where the value must be an integer in the range **[1,32]**. Default is **4**. For optimum performance, it is best to set this value to the block size of the given operating system. On Linux operating systems, a typical block size is 4 KiB.



NOTE

Larger buffers might provide improved performance for applications with transaction rates that exceed 5K transactions per second and a large number of threads.

aries.transaction.howl.logFileDir

(Required) Specifies the log directory location, which must be an absolute path. No default value.

aries.transaction.howl.logFileName

Specifies the name of the HOWL log file. Default is **transaction**.

aries.transaction.howl.logFileExt

Specifies the file extention for the HOWL log file. Default is **log**.

aries.transaction.howl.maxLogFiles

Specifies the maximum number of log files. Default is 2.

aries.transaction.howl.maxBlocksPerFile

Specifies the maximum size of a transaction log file in blocks, where the block size is defined by the **aries.transaction.howl.bufferSize** property. After the maximum size is reached, the log rolls over to a new log file. Default is **-1** (that is, **0x7fffffff** blocks).

Sample settings

The following example shows the default settings from the **org.apache.aries.transaction.cfg** configuration file:

```
aries.transaction.timeout=600
aries.transaction.howl.logFileDir=${karaf.data}/txlog/
```

6.3. ACCESSING THE TRANSACTION MANAGER

Overview

The easiest way for an application to access the Aries transaction manager inside the OSGi container is to create a bean reference to the OSGi service using Blueprint XML. In fact, you typically need to reference two OSGi services: the JTA transaction manager and the Spring **PlatformTransactionManager**. These two services access the *same* underlying transaction manager, but use alternative wrapper layers (see Figure 6.1, “OSGi Transaction Architecture”).

Blueprint XML

In blueprint XML, you can get bean references to the JTA transaction manager instance and to the Spring **PlatformTransactionManager** instance using the following sample code:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           default-activation="lazy">

    <!--
        OSGi TM Service
    -->
    <!-- access through Spring's PlatformTransactionManager -->
    <reference id="osgiPlatformTransactionManager"
               interface="org.springframework.transaction.PlatformTransactionManager"/>
    <!-- access through PlatformTransactionManager -->
    <reference id="osgiJtaTransactionManager"
               interface="javax.transaction.TransactionManager"/>

</blueprint>
```

6.4. JAVA TRANSACTION API

Overview

The Red Hat JBoss Fuse transaction implementation can be accessed through the standard JTA interfaces (for example, **javax.transaction.UserTransaction** and

javax.transaction.TransactionManager). In practice, it is rarely necessary to access the these JTA interfaces directly, because Red Hat JBoss Fuse provides alternative ways of accessing and managing transactions.

UserTransaction interface

The JTA **UserTransaction** interface has the following definition:

```
public interface javax.transaction.UserTransaction {
    public void begin();

public void commit();
    public void rollback();

    public void setRollbackOnly();


public int getStatus();

    public void setTransactionTimeout(int seconds);
}
```

UserTransaction methods

The **UserTransaction** interface defines the following methods:

begin()

Start a new transaction, associating it with the current thread. If any XA resources get associated with this transaction, it implicitly becomes an XA transaction.

commit()

Complete the current transaction normally, so that all pending changes become permanent. After the commit, there is no longer a transaction associated with the current thread.

NOTE

If the current transaction is marked as *rollback only*, however, the transaction would actually be rolled back when **commit()** is called.

rollback()

Abort the transaction immediately, so that all pending changes are discarded. After the rollback, there is no longer a transaction associated with the current thread.

setRollbackOnly()

Modify the state of the current transaction, so that a rollback is the only possible outcome, but do not perform the rollback yet.

getStatus()

Returns the status of the current transaction, which can be one of the following integer values defined in the **javax.transaction.Status** interface:

- **STATUS_ACTIVE**
- **STATUS_COMMITTED**
- **STATUS_COMMITTING**
- **STATUS_MARKED_ROLLBACK**
- **STATUS_NO_TRANSACTION**
- **STATUS_PREPARED**
- **STATUS_PREPARING**
- **STATUS_ROLLEDBACK**
- **STATUS_ROLLING_BACK**
- **STATUS_UNKNOWN**

setTransactionTimeout()

Customize the timeout of the current transaction, specified in units of seconds. If the transaction does not get resolved within the specified timeout, the transaction manager will automatically roll it back.

When to use UserTransaction?

The **UserTransaction** interface is used for transaction demarcation: that is, for beginning, committing, or rolling back transactions. This is the JTA interface that you are most likely to use directly in your application code. But the **UserTransaction** interface is just *one* of the ways to demarcate transactions. For a complete discussion of the different ways to demarcate transactions, see XA Transaction Demarcation.

TransactionManager interface

The JTA **TransactionManager** interface has the following definition:

```
interface javax.transaction.TransactionManager {
    // Same as UserTransaction methods
    public void begin();

    public void commit();

    public void rollback();
    public void setRollbackOnly();

    public int getStatus();

    public void setTransactionTimeout(int seconds);

    // Extra TransactionManager methods
    public Transaction getTransaction();
    public Transaction suspend() ;
    public void resume(Transaction tobj);
}
```

TransactionManager methods

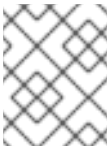
The **TransactionManager** interface supports all of the methods found in the **UserTransaction** interface (so you can also use it for transaction demarcation) and, in addition, supports the following methods:

getTransaction()

Get a reference to the current transaction (that is, the transaction associated with the current thread), if any. If there is no current transaction, this method returns **null**.

suspend()

Detach the current transaction from the current thread, returning a reference to the transaction. After calling this method, the current thread no longer has a transaction context, so that any work that you do after this point is no longer done in the context of a transaction.



NOTE

Not all transaction managers support suspending transactions. This feature is supported by Apache Geronimo, however.

resume()

Re-attach a suspended transaction to the current thread context. After calling this method, the transaction context is restored and any work that you do after this point is done in the context of a transaction.

When to use TransactionManager?

The most common way to use a **TransactionManager** object is simply to pass it to a framework API (for example, to the Camel JMS component), enabling the framework to look after transaction demarcation for you. Occasionally, you might want to use the **TransactionManager** object directly, if you need to access advanced transaction APIs (such as **suspend()** and **resume()**).

Transaction interface

The JTA **Transaction** interface has the following definition:

```
interface javax.transaction.Transaction {
    public void commit();

    public void rollback();

    public void setRollbackOnly();
    public int getStatus();

    public boolean enlistResource(XAResource xaRes);

    public boolean delistResource(XAResource xaRes, int flag);
    public void registerSynchronization(Synchronization sync);
}
```

Transaction methods

The **commit()**, **rollback()**, **setRollbackOnly()**, and **getStatus()** methods have just the same effect as the corresponding methods from the **UserTransaction** interface (in fact, the **UserTransaction** object is really just a convenient wrapper that retrieves the current transaction and then invokes the corresponding methods on the **Transaction** object).

Additionally, the **Transaction** interface defines the following methods that have no counterpart in the **UserTransaction** interface:

enlistResource()

Associate an XA resource with the current transaction.

NOTE

This method is of key importance in the context of XA transactions. It is precisely the capability to enlist multiple XA resources with the current transaction that characterizes XA transactions. On the other hand, enlisting resources explicitly is a nuisance and you would normally expect your framework or container to do this for you. For example, see Section 6.5, “The XA Enlistment Problem”.

delistResource()

Disassociates the specified resource from the transaction. The **flag** argument can take one of the following integer values defined in the **javax.transaction.Transaction** interface:

- **TMSUCCESS**
- **TMFAIL**
- **TMSUSPEND**

registerSynchronization()

Register a **javax.transaction.Synchronization** object with the current transaction. The **Synchronization** object is an object that receives a callback just before the prepare phase of a commit and a callback just after the transaction completes.

When to use Transaction?

You would only need to access a **Transaction** object directly, if you are suspending/resuming transactions or if you need to enlist a resource explicitly. As discussed in Section 6.5, “The XA Enlistment Problem”, a framework or container would usually take care of enlisting resources automatically for you.

Reference

For fulls details of the Java transaction API, see the Java Transaction API (JTA) 1.1 specification.

6.5. THE XA ENLISTMENT PROBLEM

The problem of XA enlistment

The standard JTA approach to enlisting XA resources is to add the XA resource explicitly to the current **javax.transaction.Transaction** object (representing the current transaction). In other words, you must explicitly enlist an XA resource *every time a new transaction starts*.

How to enlist an XA resource

Enlisting an XA resource with a transaction simply involves invoking the **enlistResource()** method on the Transaction interface. For example, given a **TransactionManager** object and an **XAResource** object, you could enlist the **XAResource** object as follows:

```
// Java
import javax.transaction.Transaction;
import javax.transaction.TransactionManager;
import javax.transaction.xa.XAResource;
...
// Given:
// 'tm' of type TransactionManager
// 'xaResource' of type XAResource

// Start the transaction
tm.begin();

Transaction transaction = tm.getTransaction();
transaction.enlistResource(xaResource);

// Do some work...
...

// End the transaction
tm.commit();
```

Auto-enlistment

The tricky aspect of enlisting resources is that the resource must be enlisted on *every* new transaction and the resource must be enlisted *before* you start to use the resource. If you enlist resources explicitly, you could end up with error-prone code that is littered with **enlistResource()** calls. Moreover, sometimes it can be difficult to call **enlistResource()** in the right place (for example, if you are using a framework that hides some of the transaction details).

For these reasons, it is much easier (and safer) in practice to use features that support *auto-enlistment* of XA resources. For example, in the context of using JMS and JDBC resources, the standard technique is to use *wrapper classes* that support auto-enlistment.

JMS XA wrapper

The way to perform auto-enlisting of a JMS XA resource is to implement a wrapper class of type, **javax.jms.ConnectionFactory**. You can then implement this class, so that every time a new connection is created, the JMS XA resource is enlisted with the current transaction.

Apache ActiveMQ provides the following auto-enlisting wrapper classes:

XaPooledConnectionFactory

A generic XA pooled connection factory that automatically enlists the XA resource for each transaction and pools JMS connections, sessions and message producers.

JcaPooledConnectionFactory

An XA pooled connection factory that works with the Geronimo transaction manager (Aries) and provides for proper recovery after a system or application failure.

To use the **JcaPooledConnectionFactory** wrapper class, create an instance that takes a reference to an XA connection factory instance (for example, **ActiveMQXAConnectionFactory**), provide a reference to the transaction manager (which is used to enlist the resource), and specify a unique name for this XA resource (needed to support XA recovery).

For example, the following example shows how you can use Blueprint XML to define an auto-enlisting JMS connection factory (with the bean ID, **jmsXaPoolConnectionFactory**):

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  ...
  <!-- access through JTA TransactionManager -->
  <reference id="osgiJtaTransactionManager"
             interface="javax.transaction.TransactionManager"/>
  ...
  <!-- connection factory wrapper to support auto-enlisting of XA resource -->
  <bean id="jmsXaPoolConnectionFactory"
        class="org.apache.activemq.pool.JcaPooledConnectionFactory">
    <property name="name" value="MyXaResourceName" />
    <property name="maxConnections" value="1" />
    <property name="connectionFactory" ref="jmsXaConnectionFactory" />
    <property name="transactionManager" ref="osgiJtaTransactionManager" />
  </bean>

  <bean id="jmsXaConnectionFactory"
        class="org.apache.activemq.ActiveMQXAConnectionFactory">
    <property name="brokerURL" value="vm:local"/>
    <property name="userName" value="UserName"/>
    <property name="password" value="Password"/>
    <property name="redeliveryPolicy">
      <bean class="org.apache.activemq.RedeliveryPolicy">
        <property name="maximumRedeliveries" value="0"/>
      </bean>
    </property>
  </bean>
  ...
</blueprint>
```

6.6. GENERIC XA-AWARE CONNECTION POOL LIBRARY

Overview

The generic XA-aware connection pool library is provided via the **org.apache.activemq.jms.pool** component. The library enables third-party JMS providers to participate in XA transactions managed by any JTA transaction manager—Apache Geronimo in particular—and to recover properly from system and application failures.

When an application uses the JMS bridge or the camel JMS component with a third-party JMS provider, this library enables it to leverage the shared pooled implementation at both ends of the bridge or the camel route.

NOTE

org.apache.activemq.pool extends **org.apache.activemq.jms.pool**.

The library contains three different pooled connection factories:

- **PooledConnectionFactory**

  PooledConnectionFactory is a simple generic connection factory that pools connection, session, and message producer instances, so they can be used with tools, such as Apache Camel and Spring's JMSTemplate and MessageListenerContainer. Connections, sessions, and producers are returned to the pool for later reuse by the application.

- **XaPooledConnectionFactory**

  XaPooledConnectionFactory is a generic XA pooled connection factory that automatically enlists the XA resource for each transaction and pools JMS connections, sessions and message producers.

- **JcaPooledConnectionFactory**

  JcaPooledConnectionFactory is a generic XA pooled connection factory that works with the Geronimo transaction manager (Aries) and provides for proper recovery after a system or application failure using the **GenericResourceManager**. It also automatically enlists the XA resource for each transaction and pools JMS connections, sessions and message producers.

Dependencies

Maven users need to add a dependency on **activemq-jms-pool** to their **pom.xml** file as shown here:

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-jms-pool</artifactId>
  <version>x.x.x</version>  <!-- Use same version as your activemq core  -->
<dependency>
```

PooledConnectionFactory

The **PooledConnectionFactory** has one required and nine optional parameters:

Table 6.1. PooledConnectionFactory parameters

Parameter	Required	Description
connectionFactory	Yes	Specifies the underlying JMS connectionFactory.

Parameter	Required	Description
blockIfSessionPoolIsFull	No	Controls the behavior of the internal session pool when it is full. • *true*—(default) Blocks until a session object becomes available • *false*—Throws an exception **Note**: **maximumActiveSessionPerConnection** controls the size of the session pool.
createConnectionOnStartup		Specifies whether a connection will be created immediately on a call to **start()**. This is useful for warming up the pool on startup. **Note**: Any exception that occurs during startup is logged at **WARN** level and ignored. Defaults to *true*.
expiryTimeout	No	Specifies the time, in milliseconds, at which connections are allowed to expire, regardless of load or ***idleTimeout***. Provides a simple way to achieve load balancing when used with the failover protocol (see Failover in the Transport Reference). Since failover works only when a connection is initially made, a pooled connection doesn't get another chance to load balance, unless expiry is forced on the pool. If your application has no need to load balance frequently—for example, when a destination's producers and consumers are colocated on the same broker—set this parameter to a large number. Defaults to *0* (disabled).

Parameter	Required	Description
idleTimeout	No	Specifies the time, in milliseconds, newly created connections can remain idle before they are allowed to expire. On expiring, a connection is closed and removed from the pool. **Note**: Connections are normally tested on each attempt to borrow one from the pool. So if connections are infrequently requested, a connection instance could remain in the pool much longer than its configured setting. Defaults to *30*1000* .
maxConnections	No	Specifies the maximum number of connections to use. Each call to **createConnection** creates a new connection, up to ***maxConnections***. Defaults to *1*.
maximumActiveSessionPerConnection	No	Specifies the maximum number of active sessions allowed per connection. Once this maximum has been reached and a new session is requested, the connection either throws an exception or blocks until a session becomes available, according to **blockIfSessionPoolIsFull**. Defaults to *500*.
numConnections	No	Specifies the number of connections currently in the pool.

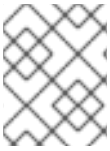
Parameter	Required	Description
timeBetweenExpirationCheckMillis	No	Specifies the number of milliseconds to sleep between each run of the idle connection eviction thread. • **non positive value**—no eviction thread is run. Pooled connections are checked for excessive idle time or other failures only when one is borrowed from the pool. • **positive value**—Specifies the number of milliseconds to wait between each run of the idle connection eviction thread. Defaults to *-1*, so no eviction thread will ever run.
useAnonymousProducers	No	Specifies whether sessions will use one anonymous MessageProducer for all producer requests or create a new MessageProducer for each producer request. • *true*—(default) Only one anonymous MessageProducer is allocated for all requests You can use the **MessageProducer#send(destination, message)** method with anonymous message producers. **Note**: Using this approach prevents the broker from showing producers per destination. • *false*—A new MessageProducer is allocated per request

XaPooledConnectionFactory

The **XaPooledConnectionFactory** extends the **PooledConnectionFactory**, implementing three additional optional parameters:

Table 6.2. XaPooledConnectionFactory parameters

Parameter	Required	Description
transactionManager	No	Specifies the JTA transaction manager to use.
tmFromJndi	No	Resolves a transaction manager from JNDI using the specified **tmJndiName**.
tmJndiName	No	Specifies the name of the transaction manager to use when resolving one from JNDI. Defaults to **java:/TransactionManager**.



NOTE

When no transaction manager is specified, the **XaConnecionPool** class reverts to behaving as a non XA-based connection.

JcaPooledConnectionFactory

The **JcaPooledConnectionFactory** pool extends the**XaPooledConnectionFactory**, implementing one additional required parameter:

Table 6.3. JcaPooledConnectionFactory parameters

Parameter	Required	Description
name	Yes	Specifies the name of the resource manager that the JTA transaction manager will use to detect whether two-phase commits must be employed. The resource manager name must uniquely identify the broker. **Note:** To start the recovery process, the **GenericResourceManager** must also be configured.

Examples

- PooledConnectionFactory

  This example (Example 6.1) shows a simple pool that configures some connection parameters for a standalone ActiveMQ broker. In this scenario, you can use either the activemq-specific pool **org.apache.activemq.pool** or the generic pool **org.apache.activemq.jms.pool**.

  Example 6.1. Simple pooled connection factory configured using Blueprint

```
<bean id="internalConnectionFactory"
      class="org.apache.activemq.ConnectionFactory">
  <argument value="tcp://localhost:61616" />
</bean>

<bean id="connectionFactory"
      class="org.apache.activemq.jms.pool.PooledConnectionFactory"
      init-method="start" destroy-method="stop">
       <property name="connectionFactory"  ref="internalConnectionFactory"/>
       <property name="name" value="activemq" />
       <property name="maxConnections" value="2" />
       <property name="blockIfSessionPoolIsFull" value="true" />
</bean>
```

- XaPooledConnectionFactory

  This example (Example 6.2) uses two data sources, one standalone ActiveMQ broker and one standalone HornetMQ broker, to perform XA transactions, in which data is consumed from the HornetMQ broker and produced to the ActiveMQ broker. The HornetMQ broker is configured to use the generic pool **org.apache.activemq.jms.pool**, while the ActiveMQ broker is configured to use the activemq-specific pool **org.apache.activemq.pool**. This example uses the default settings for the optional parameters.

Example 6.2. XA pooled connection factory configured programmatically

```
//Transaction manager
   javax.transaction.TransactionManager transactionManager;

//generic pool used for hornetq
 org.apache.activemq.jms.pool.XaPooledConnectionFactory
 hqPooledConnectionFactory=new
org.apache.activemq.jms.pool.XaPooledConnectionFactory();

//pooledConnectionFactory for activemQ
  XaPooledConnectionFactory amqPooledConnectionFactory=new
XaPooledConnectionFactory();

//set transaction manager
  hqPooledConnectionFactory.setTransactionManager(transactionManager);
  amqPooledConnectionFactory.setTransactionManager(transactionManager);

//set connection factory
  amqPooledConnectionFactory.setConnectionFactory(new
ActiveMQXAConnectionFactory
     ("admin","admin", "tcp://localhost:61616"));
  hqPooledConnectionFactory.setConnectionFactory(getHornetQConnectionFactory());

//create Connections
  XAConnection hornetQXaConnection=(XAConnection)
    ((org.apache.activemq.jms.pool.PooledConnection)
       hqPooledConnectionFactory.createConnection()).getConnection();
  XAConnection amqXAConnection=(XAConnection)
    ((org.apache.activemq.jms.pool.PooledConnection)
       amqPooledConnectionFactory.createConnection()).getConnection();
```

```
    hornetPooledConn=hqPooledConnectionFactory.createConnection();
    amqPooledConnection=amqPooledConnectionFactory.createConnection();

    hornetQXaConnection.start();
    amqXAConnection.start();

    transactionManager.begin();
    Transaction trans=transactionManager.getTransaction();

//XA resources are automatically enlisted by creating session
  hornetQXasession=(XASession) hornetPooledConn.createSession
    (false, Session.AUTO_ACKNOWLEDGE);
  amqXASession=(XASession) amqPooledConnection.createSession
    (false,Session.AUTO_ACKNOWLEDGE);

//some transaction ....
  trans.rollback();

//enlist again ..
  hornetQXasession=(XASession)
hornetPooledConn.createSession(false,Session.AUTO_ACKNOWLEDGE);
  amqXASession=(XASession)
amqPooledConnection.createSession(false,Session.AUTO_ACKNOWLEDGE);

//other transaction
  trans.commit();
```

- JcaPooledConnectionFactory

  This example (Example 6.3) shows the configuration of an XA application that uses the **JcaPooledConnectionFactory**, allowing a remote third-party JMS broker to participate in XA transactions with an ActiveMQ broker deployed in JBoss Fuse.

  A class specific to the Apache Geronimo transaction manager is used to register the pool with the transaction as required to enable recovery via the **GenericResourceManager**. Both the **transactionManager** and XAConnectionFactory (**ActiveMQXAConnectionFactory**) are defined and passed as properties to **JcaPooledConnectionFactory**, and the **GenericResourceManager** is configured to recover transactions upon a system or application failure.

  Example 6.3. JCA pooled connection factory configured using Blueprint

```
<reference id="transactionManager"interface="org.apache.geronimo.
        transaction.manager.RecoverableTransactionManager" availability="mandatory">

<bean id="platformTransactionManager"
      class="org.springframework.transaction.jta.JtaTransactionManager"
            init-method="afterPropertiesSet">
        <property name="transactionManager" ref="transactionManager"/>
        <property name="autodetectUserTransaction" value="false"/>
</bean>

<bean id="internalConnectionFactory"
```

```
        class="org.apache.activemq.ActiveMQXAConnectionFactory">
          <argument value="tcp://localhost:61616" />
          <property name="userName" value="admin" />
          <property name="password" value="admin" />
</bean>

<bean id="connectionFactory"
        class="org.apache.activemq.jms.pool.JcaPooledConnectionFactory"
              init-method="start" destroy-method="stop">
          <property name="connectionFactory" ref="internalConnectionFactory"/>
          <property name="transactionManager" ref="transactionManager"/>
          <property name="name" value="activemq" />
</bean>

<bean id="resourceManager"
class="org.apache.activemq.jms.pool.GenericResourceManager"
              init-method="recoverResource">
          <property name="connectionFactory" ref="internalConnectionFactory"/>
          <property name="transactionManager" ref="transactionManager"/>
          <property name="resourceName" value="activemq" />
          <property name="userName" value="admin" />
          <property name="password" value="admin" />
</bean>
```

CHAPTER 7. JMS XA TRANSACTION INTEGRATION

Abstract

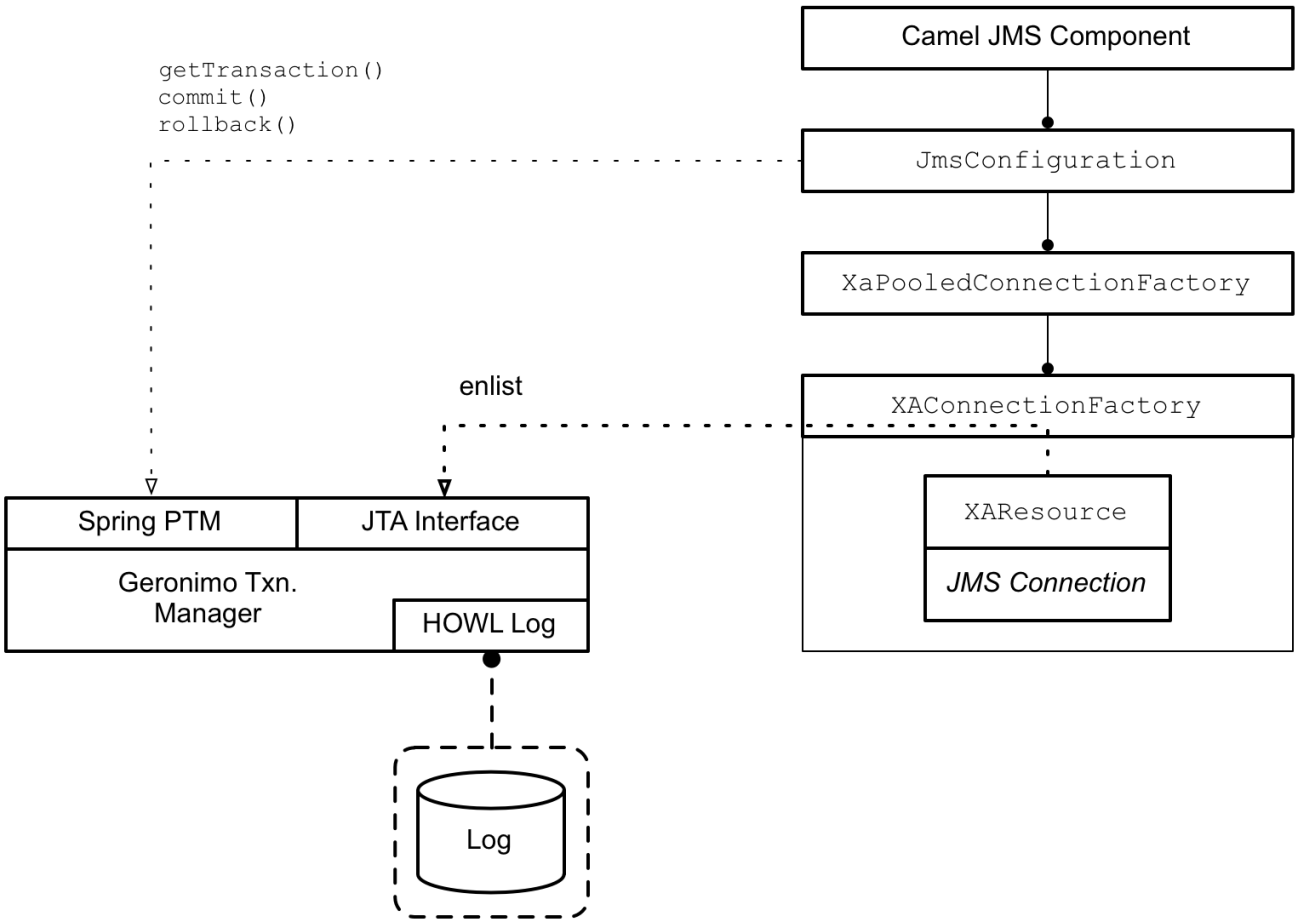
Apache ActiveMQ provides full support for XA transactions, where the broker can either be embedded in the application or deployed remotely. XA connections can be obtained through the standard **javax.jms.XAConnectionFactory** interface and Apache ActiveMQ also provides a wrapper class, which supports auto-enlistment of the JMS XA resource and connection pooling. In particular, this chapter describes in detail how to configure a Camel JMS component to use XA transactions in the context of an OSGi container.

7.1. ENABLING XA ON THE CAMEL JMS COMPONENT

Overview

Figure 7.1, “Camel JMS Component Integrated with XA Transactions” shows an overview of how to configure the Camel JMS component to use XA transactions in the Red Hat JBoss Fuse OSGi container. In this scenario, the Camel JMS component is integrated with the built-in Aries transaction manager and a connection factory wrapper is included, to support auto-enlisting of XA resources.

Figure 7.1. Camel JMS Component Integrated with XA Transactions



Accessing the XA transaction manager

The XA transaction manager, which is embedded in the OSGi container, must be accessed through two different interfaces:

org.springframework.transaction.PlatformTransactionManager

The **PlatformTransactionManager** interface is needed by the Camel JMS component (which is layered over the Spring transaction API). For more details, see PlatformTransactionManager Interface.

javax.transaction.TransactionManager

The **TransactionManager** interface is needed by the XA pooled connection factory, which uses it to enlist the ActiveMQ XA resource.

org.apache.geronimo.transaction.manager.RecoverableTransactionManager

The **RecoverableTransactionManager** interface is inherited from **javax.transaction.TransactionManager** and is needed to define a recoverable resource. For more details, see Sample JMS XA Configuration.

The transaction manager interfaces are accessed as OSGi services. For example, to access the interfaces through Blueprint XML, you can use the following code:

```
<beans ...>

  <!--
    OSGi TM Service
  -->
  <!-- access through Spring's PlatformTransactionManager -->
  <reference id="osgiPlatformTransactionManager"
          interface="org.springframework.transaction.PlatformTransactionManager">
  <!-- access through JTA TransactionManager -->
  <reference id="osgiJtaTransactionManager"
          interface="javax.transaction.TransactionManager">
  <reference id="recoverableTxManager"
    interface="org.apache.geronimo.transaction.manager.RecoverableTransactionManager">

</beans>
```

For more details, see Accessing the Transaction Manager.

XA connection factory bean

The basic connection factory bean for Apache ActiveMQ is an instance of the class, **ActiveMQXAConnectionFactory**, which exposes the **javax.jms.XAConnectionFactory** interface. Through the JTA **XAConnectionFactory** interface, it is possible to obtain a reference to an **XAResource** object, but the basic connection factory bean does *not* have the capability to auto-enlist the XA resource.

XA pooled connection factory bean

The XA pooled connection factory bean, which can be an instance of **JcaPooledConnectionFactory** type or **XaPooledConnectionFactory** type, is a connection factory wrapper class that adds the following capabilities to the basic connection factory:

- *JMS connection pooling*—enables the re-use of JMS connection instances. When a transaction is completed, the corresponding JMS connection can be returned to a pool and then re-used by another transaction.

- *Auto-enlistment of XA resources*—the pooled connection factory bean also has the ability to enlist an XA resource automatically, each time a transaction is started.

The **JcaPooledConnectionFactory** bean exposes the standard **javax.jms.ConnectionFactory** interface (but *not* the **XAConnectionFactory** interface).

Camel JMS component and JMS configuration bean

The JMS configuration bean encapsulates all of the required settings for the Camel JMS component. In particular, the JMS configuration bean includes a reference to the transaction manager (of **PlatformTransactionManager** type) and a reference to the XA pooled connection factory (of **JcaPooledConnectionFactory** type).

The **org.apache.camel.component.jms.JmsConfiguration** class supports the following bean properties, which are particularly relevant to transactions:

transacted

*Must be set to **false** for XA transactions.* The name of this property is misleading. What it really indicates is whether or not the Camel JMS component supports *local transactions*. For XA transactions, on the other hand, you must set this property to **false** and initialize the **transactionManager** property with a reference to an XA transaction manager.

This property gets its name from the **sessionTransacted** property in the underlying Spring transaction layer. The **transacted** property ultimately gets injected into the **sessionTransacted** property in the Spring transaction layer, so it is the Spring transaction layer that determines the semantics. For more details, see the Javadoc for Spring's AbstractMessageListenerContainer.

transactionManager

Must be initialized with a reference to the **PlatformTransactionManager** interface of the built-in OSGi transaction manager.

transactionName

Sets the transaction name. Default is **JmsConsumer[*destinationName*]**.

cacheLevelName

Try setting this initially to **CACHE_CONNECTION**, because this will give you the best performance. If this setting turns out to be incompatible with your transaction system, you can revert to **CACHE_NONE**, whcih switches off all caching in the Spring transaction layer. For more details, see the Spring documentation.

transactionTimeout

Do *not* set this property in the context of XA transactions. To customize the transaction timeout in the context of XA transactions, you need to configure the timeout directly in the OSGi transaction manager instead (see Configuring the Transaction Manager for details).

lazyCreateTransactionManager

Do *not* set this boolean property in the context of XA transactions. (In the context of a local transaction, setting this property to **true** would direct the Spring transaction layer to create its own local transaction manager instance, whenever it is needed.)

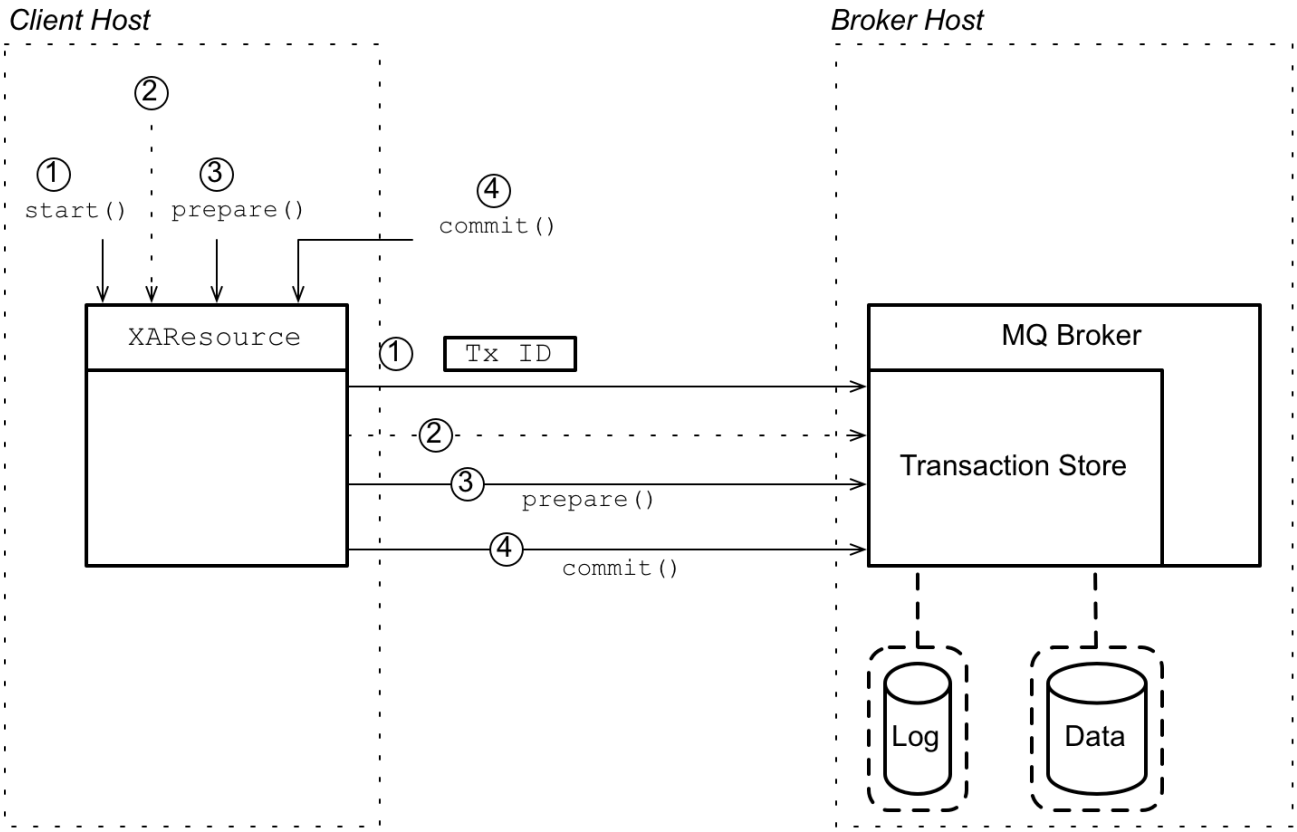
7.2. JMS XA RESOURCE

Overview

The Apache ActiveMQ implementation of the JMS XA resource supports the two-phase commit protocol and also implicitly supports distributed transactions. To understand how this works, we take a closer look at the interaction between a JMS XA resource and the broker to which it is connected.

Figure 7.2, “JMS XA Resource Connected to Remote Broker” illustrates what happens when a JMS XA resource participates in a transaction that is committed using the two-phase commit protocol. The JMS XA resource is deployed in an application that runs in the **Client** host and the corresponding Apache ActiveMQ broker is deployed on the **Remote** host, so that this transaction branch is effectively a distributed transaction.

Figure 7.2. JMS XA Resource Connected to Remote Broker



XA two-phase commit process

The two-phase commit shown in Figure 7.2, “JMS XA Resource Connected to Remote Broker” consists of the following steps:

1. Immediately after the transaction begins, the transaction manager invokes **start()** on the JMS XA resource, which indicates that the resource should initialize a new transaction. The JMS XA resource now generates a new transaction ID and sends it over the network to the remote broker.

2. The JMS XA resource now forwards all of the operations that arise during a JMS session (for example, messages, acknowledgments, and so on) to the remote broker.

   On the broker side, the received operations are *not* performed immediately. Because the operations are happening in a transaction context and the transaction is not yet committed, the broker buffers all of the operations in a *transaction store* (held in memory, initially). Messages held in the transaction store are *not* forwarded to JMS consumers.

3. In a two-phase commit process, the first phase of completing the transaction is where the transaction manager invokes **prepare()** on all of the participating XA resources. At this stage, the JMS XA resource sends the **prepare()** operation to the remote broker.

   On the broker side, when the transaction store receives the **prepare()** operation, *it writes all of the buffered operations to disk*. Hence, after the prepare phase, there is no longer any risk of losing data associated with this transaction branch.

4. The second phase of completing the transaction is where the transaction manager invokes **commit()** on all of the participating XA resources. The JMS XA resource sends the **commit()** operation to the remote broker.

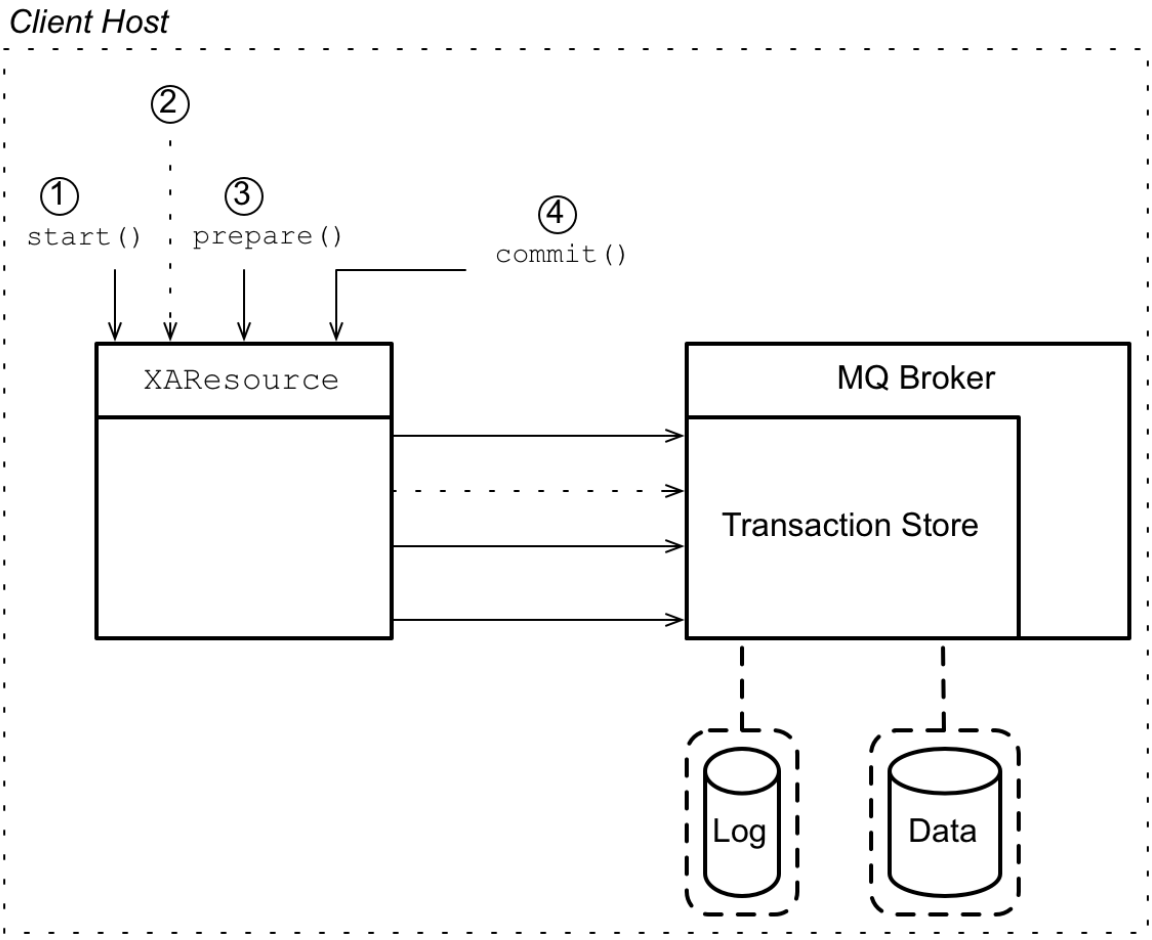
   On the broker side, the transaction store marks this transaction as complete. The pending operations are now executed and any pending messages can now be forwarded to JMS consumers.

Embedded MQ broker

Although Apache ActiveMQ supports a remote connection between the JMS XA resource and the broker, this is not the most efficient or reliable way to set up a transactional application. A network connection usually introduces significant latency and any communication delays between the JMS XA resource and the broker would affect all of the other participants in the transaction.

A more efficient approach would be to embed a broker in the same JVM as the JMS XA resource on the **Client** host, as shown in Figure 7.3, “JMS XA Resource Connected to Embedded Broker”. In this scenario, an additional broker is deployed on the **Client** host, preferably running in the same JVM as the JMS XA resource (that is, in embedded mode). Now all of the resource-to-broker communication is localized and runs much faster. It still might be necessary to forward messages to a remote broker, but this communication has no effect on the XA transactions.

Figure 7.3. JMS XA Resource Connected to Embedded Broker



Default MQ broker

By default, a standalone JBoss Fuse container already has an MQ broker deployed in it. If you deploy a JMS XA resource into this container, you can communicate efficiently with the default broker through the JVM, by connecting through the broker URL, **vm:amq**.

7.3. SAMPLE JMS XA CONFIGURATION

Spring XML configuration

Example 7.1, “Camel JMS Component with XA Enabled” shows the complete Blueprint XML configuration required to initialize the **jmstx** Camel JMS component with XA transactions. After setting up the Camel JMS component with this code, you can create a transactional JMS endpoint in a route using a URL like **jmstx:queue:*QueueName***.

Example 7.1. Camel JMS Component with XA Enabled

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <!--
       OSGi TM Service
    -->
    <!-- access through Spring's PlatformTransactionManager -->
```

```
<reference id="osgiPlatformTransactionManager"
1       interface="org.springframework.transaction.PlatformTransactionManager">
    <!-- access through PlatformTransactionManager -->
<reference id="recoverableTxManager"
    interface="org.apache.geronimo.transaction.manager.RecoverableTransactionManager"
2 availability="mandatory">
    ...
<!--
    JMS TX endpoint configuration
-->
<bean id="jmstx"
3       class="org.apache.activemq.camel.component.ActiveMQComponent">
      <property name="configuration" ref="jmsTxConfig">
</bean>

<bean id="jmsTxConfig"
4       class="org.apache.camel.component.jms.JmsConfiguration">
      <property name="connectionFactory" ref="jmsXaPoolConnectionFactory">
    <property name="transactionManager" ref="osgiPlatformTransactionManager">
    <property name="transacted" value="false">
    <property name="cacheLevelName" value="CACHE_CONNECTION">
</bean>

<!-- connection factory wrapper to support auto-enlisting of XA resource -->
<bean id="jmsXaPoolConnectionFactory"
5       class="org.apache.activemq.pool.JcaPooledConnectionFactory">
      <property name="name" value="MyXaResourceName">
6     <property name="maxConnections" value="1">
      <property name="connectionFactory" ref="jmsXaConnectionFactory">
    <property name="transactionManager" ref="recoverableTxManager">
</bean>

<bean id="jmsXaConnectionFactory"
7       class="org.apache.activemq.ActiveMQXAConnectionFactory">
8       <property name="brokerURL" value="vm:local">
        <property name="userName" value="UserName">
    <property name="password" value="Password">
9     <property name="redeliveryPolicy">
        <bean class="org.apache.activemq.RedeliveryPolicy">
       <property name="maximumRedeliveries" value="0">
      </bean>
    </property>
</bean>

<!--
    ActiveMQ XA Resource Manager
-->
<bean id="resourceManager"
     class="org.apache.activemq.pool.ActiveMQResourceManager"
     init-method="recoverResource">
    <property name="transactionManager" ref="recoverableTxManager">
    <property name="connectionFactory" ref="jmsXaConnectionFactory">
10    <property name="resourceName" value="activemq.default">
   </bean>
...
</blueprint>
```

Listing notes

The preceding Spring XML configuration can be explained as follows:

1. Define a reference to the OSGi service that exposes the **PlatformTransactionManager** interface of the OSGi container's built-in XA transaction manager. This service can then be accessed through the bean ID, **osgiPlatformTransactionManager**.

2. Define a reference to the OSGi service that exposes the recoverable **TransactionManager** interface of the OSGi container's built-in XA transaction manager. This service can then be accessed through the bean ID, **recoverableTxManager**.

3. The bean identified by the ID, **jmstx**, is the ActiveMQ implementation of the Camel JMS component. You can use this component to define transactional JMS endpoints in your routes. The only property that you need to set on this bean is a reference to the **JmsConfiguration** bean with the ID, **jmsTxConfig**.

4. The **JmsConfiguration** bean with the ID, **jmsTxConfig**, is configured as described in Section 7.1, "Enabling XA on the Camel JMS Component". In particular, the configuration bean gets a reference to the XA pooled connection factory and a reference to the **osgiPlatformTransactionManager** bean. The transacted property *must* be set to **false**.

5. The **JcaPooledConnectionFactory** is a wrapper class that adds extra capabilities to the basic connection factory bean (that is, it adds the capabilities to auto-enlist XA resources and to pool JMS connections).

6. The **maxConnections** property should be set to 1.

7. The bean with the ID, **jmsXaConnectionFactory**, is the basic connection factory, which encapsulates the code for connecting to the JMS broker. In this case, the bean is an instance of **ActiveMQXAConnectionFactory** type, which is a special connection factory class that you must use when you want to connect to the ActiveMQ broker with support for XA transactions.

8. The **brokerURL** property defines the protocol for connecting to the broker. In this case, the **vm:local** URL connects to the broker that is embedded in the current JVM and is identified by the name **local** (the configuration of the embedded broker is not shown in this example).

   There are many different protocols supported by Apache ActiveMQ that you could use here. For example, to connect to a remote broker through the OpenWire TCP protocol listening on TCP port **61616** on host **MyHost**, you would use the broker URL, **tcp://MyHost:61616**.

9. In this example, the redelivery policy is disabled by setting **maximumRedeliveries** to **0**. Typically, you would not use a redelivery policy together with transactions. An alternative approach would be to define an exception handler that routes failed exchanges to a dead letter queue.

10. The **resourceName** property is the key entry that maps from the transaction manager log to a real-world **XAResource** object. It must be unique for each **XAResource** object.

7.4. XA CLIENT WITH TWO CONNECTIONS TO A BROKER

Overview

A special case arises where an XA client opens two separate connections to the *same* remote broker instance. You might want to open two connections, for example, in order to send messages to the broker with different properties and qualities of service.

Each XA connection is implicitly associated with its own dedicated XA resource object. When two XA resource objects are equivalent (as determined by calling **XAResource.isSameRM**), however, many Transaction Managers treat these XA resource objects in a special way: when the current transaction finishes (committed or rolled back), the Transaction Manager calls **XAResource.end** *only on the first enlisted* ***XAResource*** *instance*. This creates a problem for Apache ActiveMQ, which expects **XAResource.end** to be called on every enlisted **XAResource** instance. To avoid this problem, Apache ActiveMQ provides an option which forces the Transaction Manager to call **XAResource.end** on every XA resource instance.

jms.rmIdFromConnectionId option

To cope with the scenario where an XA client opens two connections to the *same* remote broker, it is normally necessary to set the **jms.rmIdFromConnectionId** option to **true**. The effect of setting this option to **true** is that XA resource names are then based on the *connection ID*, instead of being based on the broker ID. This ensures that all connections have distinct XA resource names, even if they are connected to the same broker instance (note that every connection is associated with its own XA resource object). A side effect of setting this option is that the Transaction Manager is guaranteed to call **XAResource.end** on each of the XA resource objects.

NOTE

When you set the **jms.rmIdFromConnectionId** option to **true**, the transaction manager adopts the 2-phase commit protocol (2-PC). Hence, there is a significant overhead associated with sending messages on one connection and receiving messages on another, when transactions are enabled.

Setting rmIdFromConnectionId option on an endpoint URI

You can enable the **rmIdFromConnectionId** option by setting **jms.rmIdFromConnectionId** to **true** on an Apache ActiveMQ endpoint URI. For example, to enable this option on an OpenWire URI:

```
tcp://brokerhost:61616?jms.rmIdFromConnectionId=true
```

Setting rmIdFromConnectionId option directly on ActiveMQXAConnectionFactory

You can enable the **rmIdFromConnectionId** option directly on the **ActiveMQXAConnectionFactory** class, by invoking the **setRmIdFromConnectionId** method. For example, you can set the **rmIdFromConnectionId** option in Java, as follows:

```
// Java
ActiveMQXAConnectionFactory cf = new ActiveMQXAConnectionFactory( ... );
cf.setRmIdFromConnectionId(true);
```

And you can set the **rmIdFromConnectionId** option in XML, as follows:

```
<!--
    ActiveMQ XA Resource Manager
-->
<bean id="resourceManager"
```

```
      class="org.apache.activemq.pool.ActiveMQResourceManager"
      init-method="recoverResource">
    <property name="transactionManager" ref="recoverableTxManager">
    <property name="connectionFactory" ref="jmsXaConnectionFactory">
    <property name="resourceName" value="activemq.default">
    <property name="rmIdFromConnectionId" value="true">
</bean>
```

Example using rmIdFromConnectionId

The following example shows you how to use the **rmIdFromConnectionId** option in the context of an XA aware JMS client written in Java:

```
// Java
import org.apache.activemq.ActiveMQXAConnectionFactory

import javax.jms.XAConnection;
import javax.jms.XASession;
import javax.jms.XATopicConnection;
import javax.transaction.xa.XAException;
import javax.transaction.xa.XAResource;
import javax.transaction.xa.Xid;
...
ActiveMQXAConnectionFactory cf = new
ActiveMQXAConnectionFactory("tcp://brokerhost:61616?jms.rmIdFromConnectionId=true");
... // Configure other connection factory options (not shown)

XAConnection connection1 = (XAConnection)cf.createConnection();
XASession session1 = connection1.createXASession();
XAResource resource1 = session1.getXAResource();

XAConnection connection2 = (XAConnection)cf.createConnection();
XASession session2 = connection2.createXASession();
XAResource resource2 = session2.getXAResource();
...
// Send messages using 'connection1' AND connection2' in this thread
...
// Commit transaction => transaction manager sends xa.end() to BOTH XAResource objects
```

In this case, the XA transaction proceeds as follows:

1. Because this is an XA example, it does not show any explicit transaction demarcation (for example, **begin** or **commit** invocations). In this case, the XA Transaction Manager (TM) is responsible for transaction demarcation. For example, if you were deploying this code into a container that supports transactions, the container would normally be responsible for transaction demarcation.

2. When you create the first **XAConnection** object, **connection1**, it automatically creates the associated **XAResource** object for this connection, **resource1**. The TM automatically enlists **resource1** into the current transaction by calling XAResource.start().

3. When you create the second **XAConnection** object, **connection2**, it automatically creates the associated **XAResource** object for this connection, **resource2**. The TM automatically *joins* **resource2** to the current transaction: the TM does this by calling XAResource.start() with the TMJOIN flag.

4. Because you have set **rmIdFromConnectionId** to **true** in this example, **resource1** and **resource2** have *different* XA resource names, which means that the TM treats them as two different resources.

5. You can now do some work in the current transaction by sending messages on **connection1** and on **connection2**. All of these message sends belong to the current transaction.

6. When the current transaction is finished (committed or rolled back), the TM will call XAResource.end() on *both* **resource1** and **resource2**. This behaviour is guaranteed, because the TM perceives **resource1** and **resource2** to be different resources (due to different XA resource names).

NOTE

If you have not set the **rmIdFromConnectionId** option, the typical behaviour of the TM at this point would be to call **XAResource.end** *only* on the first resource, **resource1**. This creates problems in the context of Apache ActiveMQ, because the second connection, **connection2**, can send messages asynchronously and these asynchronous messages will not be synchronized with the transaction unless the TM calls **XAResource.end** on **resource2** as well.

CHAPTER 8. JDBC XA TRANSACTION INTEGRATION

Abstract

In order to integrate a database with the XA transaction manager, you need two things: an XA data source (provided by the database implementation); and a proxy data source that wraps the original XA data source and supports auto-enlistment of the XA resource (provided either by the database implementation or by a third-party library). In this chapter, the JDBC integration steps are illustrated using the Apache Derby database.

8.1. CONFIGURING AN XA DATA SOURCE

Overview

A JDBC client can access an XA data source either directly, through the **javax.sql.XADataSource** interface, or indirectly, through a proxy object that implements the **javax.sql.DataSource** interface. In the context of OSGi, the usual way to integrate an XA data source is to instantiate the data source implementation provided by the underlying database and then to export that data source as an OSGi service.

javax.sql.DataSource interface

The **javax.sql.DataSource** interface is the preferred way to expose a JDBC interface. It is a highly abstracted interface, exposing only two methods, **getConnection** and **setConnection**, to the JDBC client.

According to the JDBC specification, the usual way to make a **DataSource** object available to a JDBC client is through the JNDI registry.

javax.sql.XADataSource interface

In the context of XA transactions, a JDBC data source can be exposed as a **javax.sql.XADataSource** object. The main difference between an **XADataSource** object and a plain **DataSource** object is that the **XADataSource** object returns a **javax.sql.XAConnection** object, which you can use to access and enlist an **XAResource** object.

By default, enlisting an **XAResource** object is a manual procedure. That is, when using an **XADataSource** directly, a JDBC client must explicitly write the code to obtain the **XAResource** and enlist it with the current transaction. An alternative approach is to wrap the XA data source in a proxy data source that performs enlistment automatically (for example, see Section 8.2, “Apache Aries Auto-Enlisting XA Wrapper”).

Standard JDBC data source properties

The JDBC specification mandates that a data source implementation class implements the bean properties shown in Table 8.1, “Standard DataSource Properties”. *These properties are not defined on the **javax.sql.DataSource** interface and need not all be implemented. The only required property is **description**.*

Table 8.1. Standard DataSource Properties

Property	Type	Description
databaseName	**String**	*(Optional)* Name of the database instance.
dataSourceName	**String**	*(Optional)* For an XA data source or a pooled data source, names the underlying data source object.
description	**String**	*(Required)* Description of the data source.
networkProtocol	**String**	*(Optional)* Network protocol used to communicate with the database server.
password	**String**	*(Optional)* If required, the **user** and **password** properties can be provided to open a secure connection to the database server.
portNumber	**int**	*(Optional)* TCP port number where the database server is listening.
roleName	**String**	*(Optional)* The initial SQL role name.
serverName	**String**	*(Optional)* The database server name.
user	**String**	*(Optional)* If required, the **user** and **password** properties can be provided to open a secure connection to the database server.

NOTE

Although the properties shown in this table are standardized, they are not compulsory. A given data source implementation might define some or all of the standard properties, and is also free to define additional properties not mentioned in the specification.

Apache Derby

Apache Derby is an open source database implementation, which provides a full implementation of XA transactions. In the current document, we use it as the basis for some of our examples and tutorials.

IMPORTANT

Apache Derby is neither maintained nor supported by Red Hat. No guarantees are given with respect to the robustness or correctness of its XA implementation. It is used here solely for the purposes of illustration.

Derby data sources

Apache Derby provides the following alternative data source implementations (from the **org.apache.derby.jdbc** package):

EmbeddedDataSource

A *non-XA* data source, which connects to the embedded Derby database instance identified by the **databaseName** property. If the embedded database instance is not yet running, it is automatically started in the current JVM.

EmbeddedXADataSource

An *XA* data source, which connects to the embedded Derby database instance identified by the **databaseName** property. If the embedded database instance is not yet running, it is automatically started in the current JVM.

EmbeddedConnectionPoolDataSource

A *non-XA* data source with *connection pooling* logic, which connects to the embedded Derby database instance identified by the **databaseName** property. If the embedded database instance is not yet running, it is automatically started in the current JVM.

ClientDataSource

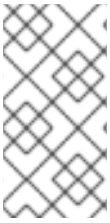
A *non-XA* data source, which connects to the remote Derby database instance identified by the **databaseName** property.

ClientXADataSource

An *XA* data source, which connects to the remote Derby database instance identified by the **databaseName** property.

ClientConnectionPoolDataSource

A *non-XA* data source with *connection pooling* logic, which connects to the remote Derby database instance identified by the **databaseName** property.



NOTE

If you need to access to the additional API methods defined in the JDBC 4.0 specification (such as **isWrapperFor**), use the variants of these data source classes with **40** appended. For example, **EmbeddedDataSource40**, **EmbeddedXADataSource40**, and so on.

Derby data source properties

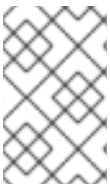
Table 8.2, “Derby DataSource Properties” shows the properties supported by the Derby data sources. For basic applications, the **databaseName** property (which specifies the database instance name) is the most important one.

Table 8.2. Derby DataSource Properties

Property	Type	Description
connectionAttributes	**String**	*(Optional)* Used to specify Derby-specific connection attributes.
createDatabase	**String**	*(Optional)* When specified with the value, **create**, the database instance specified by **databaseName** is automatically created (if it does not already exist) the next time the **getConnection** method of the data source is called
databaseName	**String**	*(Optional)* Name of the Derby database instance.
dataSourceName	**String**	*(Optional)* For an XA data source or a pooled data source, names the underlying data source object. Not used by the Derby data source implementation.
description	**String**	*(Required)* Description of the data source.
shutdownDatabase	**String**	*(Optional)* When specified with the value, **shutdown**, shuts down the database instance the next time the **getConnection** method of the data source is called.

Data sources as OSGi services

The JDBC specification recommends that data source objects are provided through the JNDI registry. In the context of the OSGi container, however, the natural mechanism for enabling loose coupling of services is the OSGi service registry. For this reason, the examples here show you how to create an XA data source and expose it as an OSGi service.



NOTE

Additionally, exposing a data source as an OSGi service has the advantage that it integrates automatically with the Aries XA data source wrapper layer. See Section 8.2, “Apache Aries Auto-Enlisting XA Wrapper”.

Blueprint

In blueprint XML, you can expose a Derby XA data source as an OSGi service using the code shown in Example 8.1, “Exposing XA DataSource as an OSGi Service in Blueprint XML”.

Example 8.1. Exposing XA DataSource as an OSGi Service in Blueprint XML

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           default-activation="lazy">

  <bean id="derbyXADataSource" class="org.apache.derby.jdbc.EmbeddedXADataSource">
    <property name="databaseName" value="txXaTutorial"/>
  </bean>

  <service ref="derbyXADataSource" interface="javax.sql.XADataSource">
   <service-properties>
     <!-- A unique ID for this XA resource. Required to enable XA recovery. -->
     <entry key="aries.xa.name" value="derbyDS"/>
     <entry key="osgi.jndi.service.name" value="jdbc/derbyXADB"/>
     <entry key="datasource.name" value="derbyXADB"/>
   </service-properties>
  </service>

</blueprint>
```

References

For more information about defining Derby data sources, see the Apache Derby manuals.

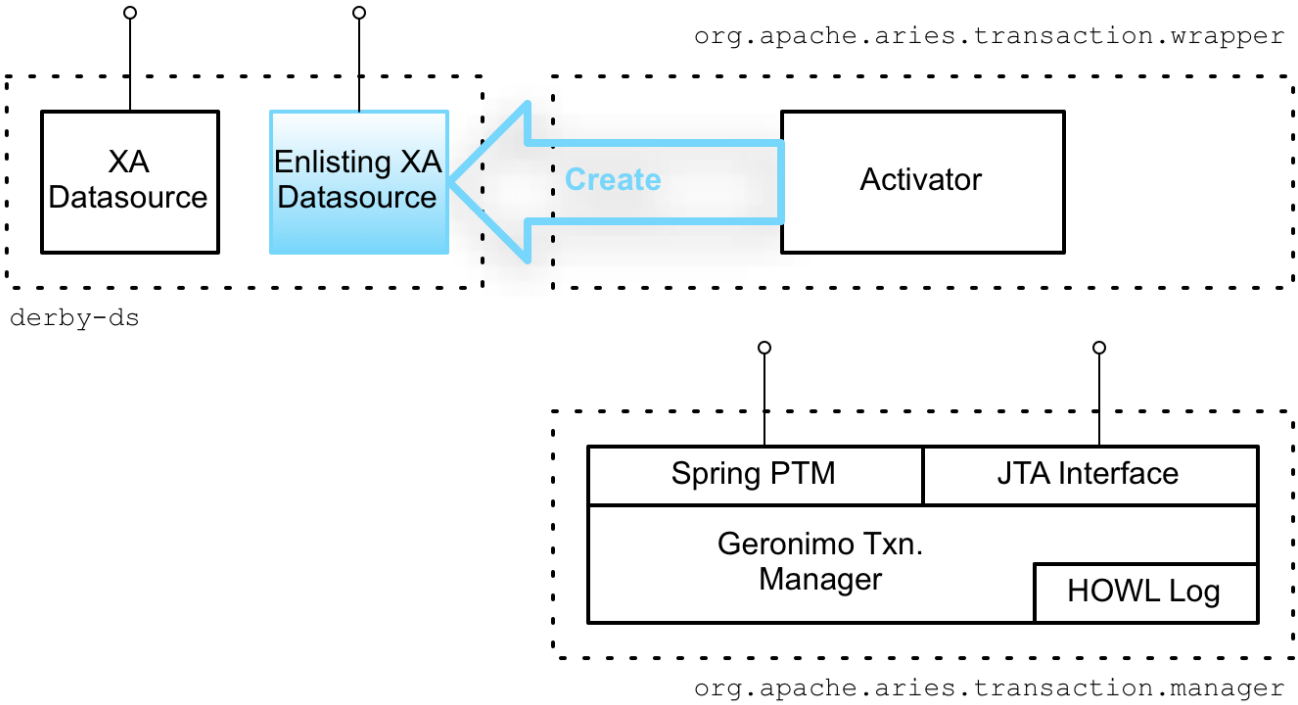
8.2. APACHE ARIES AUTO-ENLISTING XA WRAPPER

Overview

One of the features of the Apache Aries transaction module is that it provides support for auto-enlistment of XA transactions in the context of JDBC data sources. As already noted, auto-enlisting is the most practical way of integrating an XA data source with a transaction manager. The basic idea is to wrap the original data source with a *data source proxy* object that encapsulates the logic to perform auto-enlisting.

An unusual aspect of the Apache Aries' auto-enlisting feature is that the data source proxy is *automatically created for you*. In order to trigger auto-creation of the data source proxy, it is necessary to export your data source as an OSGi service. The mechanism is illustrated in Figure 8.1, “Creating the Auto-Enlisting XA Wrapper”.

Figure 8.1. Creating the Auto-Enlisting XA Wrapper



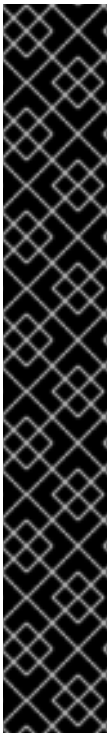
derby-ds bundle

The **derby-ds** bundle shown in Figure 8.1, “Creating the Auto-Enlisting XA Wrapper” encapsulates the code from Example 8.1, “Exposing XA DataSource as an OSGi Service in Blueprint XML”, which defines a Derby XA data source and exports it as an OSGi service.

Also shown wthin the scope of the **derby-ds** bundle is the auto-enlisting data source proxy. But this data source proxy is *not* created by the code in the **derby-ds** bundle and is initially not part of the bundle.

Automatic wrapper instantiation

Instantiation of the data source proxy depends on the Aries transaction wrapper bundle (**org.apache.aries.transaction.wrappers**). This bundle defines an *activator*, which installs hooks into the OSGi runtime, so that it is notified whenever an OSGi bundle exports a service supporting the **javax.sql.XADataSource** interface.



IMPORTANT

The Aries transaction wrapper bundle is *not* installed by default. To take advantage of the datasource wrapping functionality, you must explicitly install the **connector** feature.

To install it on a standalone JBoss Fuse container, use this command:

```
JBossFuse:karaf@root> features:install connector
```

In fabric environments, you add the **connector** feature to the profile that is deployed to the container on which the JDBC driver will enlist with the Aries transaction manager.

You can add it to the **jboss-fuse-full** profile, using this command:

```
JBossFuse:karaf@root> profile-edit --feature connector jboss-fuse-full
```

or add it to any user-defined profile, using this command:

```
JBossFuse:karaf@root> profile-edit --feature connector <custom-profile-name>
```

XADataSourceEnlistingWrapper

Upon detecting a new OSGi service supporting the **javax.sql.XADataSource** interface, the activator automatically creates a new **XADataSourceEnlistingWrapper** object, which wraps the original XA data source, effectively acting as a *data source proxy*. The **XADataSourceEnlistingWrapper** object also obtains a reference to the JTA transaction manager service (from the **org.apache.aries.transaction.manager** bundle). Finally, the activator exports the **XADataSourceEnlistingWrapper** object with the **javax.sql.DataSource** interface.

JDBC clients now have the option of accessing the XA data source through this newly created data source proxy. Whenever a new database connection is requested from the data source proxy (by calling the **getConnection** method), the proxy automatically gets a reference to the underlying XA resource and enlists the XA resource with the JTA transaction manager. This means that the required XA coding steps are automatically performed and the JDBC client does not need to be XA transaction aware.

NOTE

The **XADataSourceEnlistingWrapper** class is *not* exported from the Aries transaction wrapper bundle, so it is not possible to create the data source proxy explicitly. Instances of this class can only be created automatically by the activator in the transaction wrapper bundle.

Accessing the enlisting wrapper

If you deploy the **derby-ds** bundle, you can see how the wrapper proxy is automatically created. For example, after following the instructions in Section 10.3, “Define a Derby Datasource” and Section 10.5, “Deploy and Run the Transactional Route” to build and deploy the **derby-ds** bundle, you can list the OSGi services exported by the **derby-ds** bundle using the **osgi:ls** console command. Assuming that **derby-ds** has the bundle ID, 229, you would then enter:

```
JBossFuse:karaf@root> osgi:ls 229
```

The console produces output similar to the following:

```
-
```

```
Derby XA data source (229) provides:
------------------------------------
datasource.name = derbyXADB
objectClass = javax.sql.XADataSource
osgi.jndi.service.name = jdbc/derbyXADB
osgi.service.blueprint.compname = derbyXADataSource
service.id = 423
----
aries.xa.aware = true
aries.xa.name = derbyDS
datasource.name = derbyXADB
objectClass = javax.sql.DataSource
osgi.jndi.service.name = jdbc/derbyXADB
osgi.service.blueprint.compname = derbyXADataSource
service.id = 424
----
...
```

The following OSGi services are exposed:

- An OSGi service with interface **javax.sql.XADataSource** and **datasource.name** equal to **derbyXADB**—this is the XA data source explicitly exported as an OSGi service in Example 8.1, “Exposing XA DataSource as an OSGi Service in Blueprint XML”.
- An OSGi service with interface **javax.sql.DataSource** and **datasource.name** equal to **derbyXADB**—this is the auto-enlisting data source proxy *implicitly* created by the Aries wrapper service. The data source proxy copies the user-defined service properties from the original OSGi service and adds the setting **aries.xa.aware = true**. The **aries.xa.aware** property enables you to distinguish between the generated proxy and the original data source.

Blueprint

In blueprint XML, you can access the auto-enlisting data source proxy by defining an **reference** element as shown in Example 8.2, “Importing XA DataSource as an OSGi Service Reference in Blueprint XML”.

Example 8.2. Importing XA DataSource as an OSGi Service Reference in Blueprint XML

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           default-activation="lazy">

  <!--
     Import Derby XA data source as an OSGi service
  -->
  <reference id="derbyXADataSource"
             interface="javax.sql.DataSource"
             filter="(datasource.name=derbyXADB)"/>
</blueprint>
```

JDBC connection pool options

Additional configuration options (Table 8.3) for controlling the pooling of JDBC connections are available for a JDBC driver that is auto-enlisted in the Aries transaction manager. In the Blueprint **datasource.xml**, these options are specified as key/value pairs under the service definition's **service-properties** element.

For example, in the following Blueprint **datasource.xml** example, several of the connection pool configuration options are specified:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           default-activation="lazy">

  <bean id="derbyXADataSource" class="org.apache.derby.jdbc.EmbeddedXADataSource">
    <property name="databaseName" value="txXaTutorial"/>
  </bean>

  <service ref="derbyXADataSource" interface="javax.sql.XADataSource">
   <service-properties>
     <entry key="datasource.name" value="derbyXADB"/>
     <!-- A unique ID for this XA resource. Required to enable XA recovery. -->
     <entry key="aries.xa.name" value="derbyDS"/>
     <!-- Additional supported pool connection properties -->
     <entry key="aries.xa.pooling" value="true"/>
     <entry key="aries.xa.poolMinSize" value="1"/>
     <entry key="aries.xa.poolMaxSize" value="3"/>
     <entry key="aries.xa.partitionStrategy" value="none"/>
     <entry key="aries.xa.allConnectionsEquals" value="false"/>
   </service-properties>
  </service>

  ...

</blueprint>
```

Table 8.3. JDBC connection pool configuration options

Property	Description
aries.xa.name	Specifies the name of the managed resource that the transaction manager uses to uniquely identify and recover the resource.

Property	Description
aries.xa.exceptionSorter	(Optional) Determines whether an exception will cause the connection to be discarded and rollback of the transaction eventually attempted. Valid values are: • **all**—[Default] All exceptions considered fatal; discard the connection and attempt rollback. • **none**—No exception considered fatal; retain connection, no rollback • **known**—Only known SQL states not considered fatal; for those, retain connection/no rollback. For a list of known states, see KnownSQLStateExceptionSorter. • **custom**—Only the specified comma-separated list of SQL states not considered fatal; for those, retain connection/no rollback. For example, **<entry key="aries.xa.exceptionSorter" value="custom(06S04,09S05)" />**
aries.xa.username	(Optional) Specifies the name of the user to use. This property is usually set on the inner ConnectionFactory. However, setting it in the service definition overrides the value set in the inner ConnectionFactory.
aries.xa.password	(Optional) Specifies the password to use. This property is usually set on the inner ConnectionFactory. However, setting it also in the service definition overrides the value set in the inner ConnectionFactory.
aries.xa.pooling	(Optional) Enables/disables support for pooling. Default is **true** (enabled).
aries.xa.poolMaxSize	(Optional) Specifies the maximum pool size in number of connections. Default is **10**.
aries.xa.poolMinSize	(Optional) Specifies the minimum pool size in number of connections. Default is **0**.

Property	Description
aries.xa.transaction	(Optional) Specifies the type of transactions to use. Valid values are: • **none**—No transactions used • **xa**—[Default] XA transactions used • **local**—Non XA transactions used
aries.xa.partitionStrategy	(Optional) Specifies the pool partitioning strategy to use. Valid values are: • **none**—[Default] No partitioning • **by-subject**—Partition by authenticated users • **by-connector-properties**—Partition by connection properties
aries.xa.connectionMadIdleMinutes[a]	(Optional) Specifies the maximum time, in minutes, that a connection can remain idling before it's released from the pool. Default is **15**.
aries.xa.connectionMaxWaitMilliseconds	(Optional) Specifies the maximum time, in milliseconds, to wait to retrieve a connection from the pool. Default is **5000**.
aries.xa.allConnectionsEquals	(Optional) Specifies to assume that all connections are equal— use the same user credentials—when retrieving one from the pool. Default is **true**. **Note:** If you're using different kinds of connections to accommodate different users, do not enable this option unless you use a partition strategy that pools matching connections. Otherwise, attempts to retrieve connections from the pool will fail.
aries.xa.validateOnMatch	Specifies whether to check the validity of a connection at the time it is checked out of the connection pool. Defaults to **true**. Note that it is usually unnecessary to enable both the **validateOnMatch** option and the **backgroundValidation** option simultaneously.

Property	Description
aries.xa.backgroundValidation	Enables background validation of connections in the pool. When this option is enabled, a separate thread runs in the background to check the validity of the connections in the pool. This is typically more efficient than the **validateOnMatch** option (and it is recommended that you set the **validateOnMatch** option to **false** when this option is enabled). Defaults to **false**.
aries.xa.backgroundValidationMilliseconds	Used in combination with the **backgroundValidation** option. Specifies the interval between background validation checks in units of milliseconds. Defaults to **600000** (10 minutes).
[a] Though the spelling of this property appears incorrect, it is not. Do not replace the **d** in connection**Mad**Minutes with an **x**.	

CHAPTER 9. XA TRANSACTION DEMARCATION

Abstract

Red Hat JBoss Fuse supports a variety of different ways to demarcate XA transactions (that is, beginning, committing or rolling back transactions). You can choose the most convenient way to demarcate transactions, depending on the context.

9.1. DEMARCATION BY TRANSACTIONAL ENDPOINTS

Overview

In the context of a Apache Camel route, you have the option of enabling transaction demarcation in the consumer endpoint, which appears at the start of a route (that is, the endpoint appearing in **from(...)**). This has the advantage that the transaction scope spans the whole route, including the endpoint that starts the route. Not all endpoint types are transactional, however.

Auto-demarcation by JMS consumer endpoints

A Camel JMS consumer endpoint with XA transactions enabled will automatically demarcate a transaction as follows:

1. The endpoint automatically starts a transaction (by invoking **begin()** on the XA transaction manager), *before* pulling a message off the specified JMS queue.
2. The endpoint automatically commits the transaction (by invoking **commit()** on the XA transaction manager), after the exchange has reached the end of the route.

For example, given the XA-enabled component, **jmstx** (see Sample JMX XA Configuration), you can define a transactional route as follows:

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("jmstx:queue:giro")
          .beanRef("accountService","credit")
          .beanRef("accountService","debit");
    }
}
```

JMS producer endpoints

In contrast to consumer endpoints, JMS producer endpoints do *not* demarcate transactions (since producer endpoints typically appear at the end of a route, it would be too late to initiate a transaction anyway). Nonetheless, an XA-enabled producer endpoint is capable of participating in a transaction, if a transaction context is already present. In fact, it is essential to enable XA on a JMS producer endpoint, if you want it to participate in a transaction.

Transactional and non-transactional JMS endpoints

Because of the way that Apache ActiveMQ implements transactions, a transactional JMS endpoint must *always* be used in a transaction context and a non-transactional JMS endpoint must always be used *outside* of a transaction context. You cannot mix and match (for example, accessing a transactional JMS endpoint without any transaction context).

As a consequence of this restriction, it is typically convenient to define two different Camel JMS components, as follows:

- *A transactional Camel JMS component*—to access JMS destinations transactionally.
- *A non-transactional Camel JMS component*—to access JMS destinations without a transaction context.

9.2. DEMARCATION BY MARKING THE ROUTE

Overview

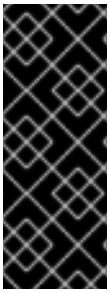
If the consumer endpoint at the start of a route does not support transactions, you can nevertheless initiate a transaction immediately after receiving an incoming message by inserting the **transacted()** command into your route.

Demarcation using transacted()

By default, the **transacted()** command uses the first transaction manager of type **PlatformTransactionManager** that it finds in the bean registry (which could either be an OSGi service, a bean defined in Spring XML, or a bean defined in blueprint). Because the **PlatformTransactionManager** interface is, by default, exported as an OSGi service, the **transacted()** command will automatically find the XA transaction manager.

When you use the **transacted()** command to mark a route as transacted, all of the processors following **transacted()** participate in the transaction; all of the processors preceding **transacted()** do *not* participate in the transaction. For example, you could use **transacted()** to make a route transactional, as follows:

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
          .transacted()
          .beanRef("accountService","credit")
          .beanRef("accountService","debit")
          .to("jmstx:queue:processed");
    }
}
```



IMPORTANT

If your container exports multiple OSGi services of **PlatformTransactionManager** type or if you register multiple **TransactedPolicy** objects in the bean registry (for example, by defining beans in Spring XML), you cannot be certain which transaction manager would be picked up by the **transacted()** command (see Default transaction manager and transacted policy). In such cases, it is recommended that you specify the transaction policy *explicitly*.

Specifying the transaction policy explicitly

To eliminate any ambiguity about which transaction manager is used, you can specify the transaction policy explicitly by passing the transaction policy's bean ID as an argument to the **transacted()** command. First of all, you need to define the transaction policy (of type, **org.apache.camel.spring.spi.SpringTransactionPolicy**), which encapsulates a reference to the transaction manager you want to use—for example:

```
<beans ...>
    ...
    <!-- access through Spring's PlatformTransactionManager -->
    <osgi:reference id="osgiPlatformTransactionManager"
                    interface="org.springframework.transaction.PlatformTransactionManager"/>
    ...
    <bean id="XA_TX_REQUIRED" class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="osgiPlatformTransactionManager"/>
    </bean>
    ...
</beans>
```

After the transaction policy bean is defined, you can use it by passing its bean ID, **XA_TX_REQUIRED**, as a string argument to the **transacted()** command—for example:

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
          .transacted("XA_TX_REQUIRED")
          .beanRef("accountService","credit")
          .beanRef("accountService","debit")
          .to("jmstx:queue:processed");
    }
}
```

For more details about transaction policies, see Propagation Policies.

XML syntax

You can also use the **transacted** command in Spring XML or blueprint files. For example, to demarcate an XA transaction in Spring XML:

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ... >

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="file:src/data?noop=true"/>
      <transacted ref="XA_TX_REQUIRED"/>
      <bean ref="accountService" method="credit"/>
      <bean ref="accountService" method="debit"/>
      <to uri="jmstx:queue:processed"/>
    </route>
  </camelContext>

</beans>
```

9.3. DEMARCATION BY USERTRANSACTION

Overview

It is possible to demarcate a transaction by accessing the **UserTransaction** service directly and calling its **begin()**, **commit()** and **rollback()** methods. But you should be careful to call these methods only when it is really necessary. Usually, in a Apache Camel application, a transaction would be started either by a transactional endpoint or by the **transacted()** marker in a route, so that explicit invocations of **UserTransaction** methods are not required.

Accessing UserTransaction from Apache Camel

In the case of Apache Camel applications deployed in an OSGi container, you can easily obtain a reference to the **UserTransaction** OSGi service by looking it up in the **CamelContext** registry. For example, given the **CamelContext** instance, **camelContext**, you could obtain a **UserTransaction** reference as follows:

```
// Java
import javax.transaction.UserTransaction;
...
UserTransaction ut =
    (UserTransaction) camelContext.getRegistry().lookup(UserTransaction.class.getName());
```

For more details of how the registry is integrated with OSGi, see Integration with Apache Camel.

Example with UserTransaction

The following example shows how to access a **UserTransaction** object and use it to demarcate a transaction, where it is assumed that this code is part of a Apache Camel application deployed inside an OSGi container.

```
// Java
import javax.transaction.UserTransaction;
...
UserTransaction ut =
    (UserTransaction) camelContext.getRegistry().lookup(UserTransaction.class.getName());

try {
  ut.begin();
```

```
  ...
  // invoke transactional methods or endpoints
  ...
  ut.commit();
}
catch (Exception e) {
  ut.rollback();
}
```

9.4. DEMARCATION BY DECLARATIVE TRANSACTIONS

Overview

You can also demarcate transactions by declaring transaction policies in your blueprint XML file. By applying the appropriate transaction policy to a bean or bean method (for example, the **Required** policy), you can ensure that a transaction is started whenever that particular bean or bean method is invoked. At the end of the bean method, the transaction will be committed. (This approach is analogous to the way that transactions are dealt with in Enterprise Java Beans).

OSGi declarative transactions enable you to define transaction policies at the following scopes in your blueprint file:

- the section called “Bean-level declaration”.
- the section called “Top-level declaration”.

Bean-level declaration

To declare transaction policies at the bean level, insert a **tx:transaction** element as a child of the **bean** element, as follows:

```
<blueprint  xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
          xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.1.0">

   <bean id="accountFoo" class="org.fusesource.example.Account">
      <tx:transaction method="*" value="Required"/>
      <property name="accountName" value="Foo"/>
   </bean>

   <bean id="accountBar" class="org.fusesource.example.Account">
      <tx:transaction method="*" value="Required"/>
      <property name="accountName" value="Bar"/>
   </bean>

</blueprint>
```

In the preceding example, the **Required** transaction policy is applied to all methods of the **accountFoo** bean and the **accountBar** bean (where the **method** attribute specifies the wildcard, *, to match all bean methods).

Top-level declaration

To declare transaction policies at the top level, insert a **tx:transaction** element as a child of the **blueprint** element, as follows:

```
<blueprint  xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
          xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.1.0">

   <tx:transaction bean="account*" value="Required"/>

   <bean id="accountFoo" class="org.fusesource.example.Account">
      <property name="accountName" value="Foo"/>
   </bean>

   <bean id="accountBar" class="org.fusesource.example.Account">
      <property name="accountName" value="Bar"/>
   </bean>

</blueprint>
```

In the preceding example, the **Required** transaction policy is applied to all methods of every bean whose ID matches the pattern, **account***.

tx:transaction attributes

The **tx:transaction** element supports the following attributes:

bean

(Top-level only) Specifies a list of bean IDs (comma or space separated) to which the transaction policy applies. For example:

```
<blueprint ...>
   <tx:transaction bean="accountFoo,accountBar" value="..."/>
</blueprint>
```

You can also use the wildcard character, *, which may appear at most once in each list entry. For example:

```
<blueprint ...>
   <tx:transaction bean="account*,jms*" value="..."/>
</blueprint>
```

If the **bean** attribute is omitted, it defaults to * (matching all non-synthetic beans in the blueprint file).

method

(Top-level and bean-level) Specifies a list of method names (comma or space separated) to which the transaction policy applies. For example:

```
<bean id="accountFoo" class="org.fusesource.example.Account">
   <tx:transaction method="debit,credit,transfer" value="Required"/>
   <property name="accountName" value="Foo"/>
</bean>
```

You can also use the wildcard character, *, which may appear at most once in each list entry.

If the **method** attribute is omitted, it defaults to * (matching all methods in the applicable beans).

value

(Top-level and bean-level) Specifies the transaction policy. The policy values have the same semantics as the policies defined in the EJB 3.0 specification, as follows:

- **Required**—support a current transaction; create a new one if none exists.
- **Mandatory**—support a current transaction; throw an exception if no current transaction exists.
- **RequiresNew**—create a new transaction, suspending the current transaction if one exists.
- **Supports**—support a current transaction; execute non-transactionally if none exists.
- **NotSupported**—do not support a current transaction; rather always execute non-transactionally.
- **Never**—do not support a current transaction; throw an exception if a current transaction exists.

CHAPTER 10. XA TUTORIAL

Abstract

This tutorial describes how to define and build a transactional route involving two XA resources (a JMS resource and a JDBC resource), based on the Apache Aries transaction manager in the OSGi container. For the purposes of illustration, the tutorial uses the Apache Derby database, which provides the JDBC XA resource.

10.1. INSTALL APACHE DERBY

Overview

For this tutorial, you need an installation of the Apache Derby database, which is an open source database that supports XA transactions. In particular, you will need to use the **ij** command-line utility later in the tutorial to create a database schema.

Downloading

Download the latest binary distribution of Apache Derby, **db-derby-*Version*-bin.zip**, from the Apache Derby download page:

> http://db.apache.org/derby/derby_downloads.html

NOTE

The same binary distribution is used both for Windows and *NIX operating systems.

Installing

To install Apache Derby, use an archive utility to extract the binary distribution into a convenient directory (for example, **C:\Program Files** on Windows, or **/usr/local** on *NIX).

Environment variables

To gain access to the Derby command-line utilities, add the Derby **bin** directory to your **PATH** variable.

On Windows, you could use a batch script like the following:

```
REM Set Apache Derby environment on Windows
SET DERBY_HOME=DerbyInstallDir

SET PATH=%DERBY_HOME%\bin;%PATH%
```

On *NIX, you could use a bash script like the following:

```
# Set Apache Derby environment on *NIX
DERBY_HOME=DerbyInstallDir

export PATH=$DERBY_HOME/bin:$PATH
```

10.2. INTEGRATE DERBY WITH JBOSS FUSE

Overview

Integrating Derby with Red Hat JBoss Fuse is a relatively simple procedure. All you need to do is to set a single Java system property, which specifies the location of the Derby system.

Derby system

A *Derby system* is essentially the directory where all of Derby's data is stored. A Derby system can contain multiple database instances, where each database instance holds its data in a sub-directory of the Derby system directory.

derby.system.home Java system property

You are required to set one Java system property to configure the Derby system: the **derby.system.home** Java system property, which specifies the location of the Derby system directory.

Setting derby.system.home in the OSGi container

In order to integrate Derby with JBoss Fuse, you must set the **derby.system.home** property in the OSGi container. Under the JBoss Fuse install directory, open the **etc/system.properties** file using a text editor and add the following line:

```
derby.system.home=DerbySystemDirectory
```

Where *DerbySystemDirectory* is any convenient location for storing your Derby data files. After setting the **derby.system.home** property, any Derby database instances created in the OSGi container will share the same Derby system (that is, the database instances will store their data under the specified Derby system directory).

10.3. DEFINE A DERBY DATASOURCE

Overview

This section explains how to define a Derby datasource (and database instance), package the datasource as an OSGi bundle, and export the datasource as an OSGi service.

Derby data source implementations

Derby provides a variety of different data source implementations, as described in Derby data sources. For XA transactions, there are two alternatives: **EmbeddedXADataSource** (where the database instance runs in the same JVM) and **ClientXADataSource** (where the application connects to a remote database instance). The current example uses **EmbeddedXADataSource**.

Auto-enlisting an XA data source

In practice, you need to wrap the basic Derby data source with an object that performs auto-enlisting of the XA data source. Apache Aries provides such a wrapper layer. In order to trigger the wrapper mechanism, however, you must export the Derby data source as an OSGi service as described in Apache Aries Auto-Enlisting XA Wrapper (and as is done in the current example).

Prerequisites

Before using Derby in the OSGi container, you must integrate Derby with the container, as described in Section 10.2, “Integrate Derby with JBoss Fuse”.

Steps to define a Derby datasource

Perform the following steps to define a Derby datasource packaged in an OSGi bundle:

1. Use the quickstart archetype to create a basic Maven project. Maven provides *archetypes*, which serve as templates for creating new projects. The Maven quickstart archetype is a basic archetype, providing the bare outline of a new Maven project.

   To create a new project for the Derby datasource bundle, invoke the Maven archetype plug-in as follows. Open a new command prompt, change directory to a convenient location (that is, to the directory where you will store your Maven projects), and enter the following command:

   ```
   mvn archetype:generate
     -DarchetypeArtifactId=maven-archetype-quickstart
     -DgroupId=org.fusesource.example
     -DartifactId=derby-ds
   ```

   NOTE

   The preceding command parameters are shown on separate lines for ease of reading. You must enter the entire command on a single line, however.

   After downloading the requisite dependencies to run the quickstart archetype, the command creates a new Maven project for the **org.fusesource.example/derby-ds** artifact under the **derby-ds** directory.

2. Change the project packaging type to **bundle**. Under the **derby-ds** directory, open the **pom.xml** file with a text editor and change the contents of the **packaging** element from **jar** to **bundle**, as shown in the following highlighted line:

   ```
   <project ...>
     ...
     <groupId>org.fusesource.example</groupId>
     <artifactId>derby-ds</artifactId>
     <version>1.0-SNAPSHOT</version>
     <packaging>bundle</packaging>
     ...
   </project>
   ```

3. Add the bundle configuration to the POM. In the **pom.xml** file, add the following **build** element as a child of the **project** element:

   ```
   <project ...>
     ...
     <build>
       <defaultGoal>install</defaultGoal>

       <plugins>
   ```

```
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>${project.groupId}.${project.artifactId}</Bundle-
SymbolicName>
          </instructions>
        </configuration>
      </plugin>

    </plugins>
  </build>

</project>
```

4. Customize the Maven compiler plug-in to enforce JDK 1.7 coding syntax. In the **pom.xml** file, add the following **plugin** element as a child of the **plugins** element, to configure the Maven compiler plug-in:

```
<project ...>
  ...
  <build>
    <defaultGoal>install</defaultGoal>

    <plugins>
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-compiler-plugin</artifactId>
       <configuration>
         <source>1.7</source>
         <target>1.7</target>
       </configuration>
     </plugin>
     ...
    </plugins>
  </build>

</project>
```

5. Add the Derby dependency to the POM and add the **derby-version** property to specify the version of Derby you are using. In the **pom.xml** file, add the **derby-version** element and the **dependency** element as shown:

```
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <derby-version>10.10.1.1</derby-version>
  </properties>

  <dependencies>
    <!-- Database dependencies -->
    <dependency>
```

```
    <groupId>org.apache.derby</groupId>
    <artifactId>derby</artifactId>
    <version>${derby-version}</version>
   </dependency>

   <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
   </dependency>
  </dependencies>
  ...
</project>
```

IMPORTANT

Remember to customize the **derby-version** property to the version of Derby you are using.

6. Instantiate the Derby database instance and export the datasource as an OSGi service. In fact, this example exports *two* datasources: an XA datasource and a non-transactional datasource. The Derby datasources are exported using a blueprint XML file, which must be stored in the standard location, **OSGI-INF/blueprint/**. Under the **derby-ds** project directory, create the **dataSource.xml** blueprint file in the following location:

```
src/main/resources/OSGI-INF/blueprint/dataSource.xml
```

Using your favorite text editor, add the following contents to the **dataSource.xml** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           default-activation="lazy">

  <bean id="derbyXADataSource" class="org.apache.derby.jdbc.EmbeddedXADataSource">
    <property name="databaseName" value="txXaTutorial">
  </bean>

  <service ref="derbyXADataSource" interface="javax.sql.XADataSource">
   <service-properties>
     <entry key="datasource.name" value="derbyXADB">
     <!-- A unique ID for this XA resource. Required to enable XA recovery. -->
     <entry key="aries.xa.name" value="derbyDS">
   </service-properties>
  </service>

  <bean id="derbyDataSource" class="org.apache.derby.jdbc.EmbeddedDataSource">
    <property name="databaseName" value="txXaTutorial">
  </bean>

  <service ref="derbyDataSource" interface="javax.sql.DataSource">
   <service-properties>
```

```
    <entry key="datasource.name" value="derbyDB">
  </service-properties>
 </service>
</blueprint>
```

In the definition of the **derbyXADataSource** bean, the **databaseName** property identifies the database instance that is created (in this case, **txXaTutorial**).

The first **service** element exports the XA datasource as an OSGi service with the interface, **javax.sql.XADataSource**. The following service properties are defined:

datasource.name

Identifies this datasource unambiguously when it is referenced from other OSGi bundles.

aries.xa.name

Defines a unique XA resource name, which is used by the Aries transaction manager to identify this JDBC resource. This property must be defined in order to support XA recovery.

The second **service** element defines a non-transactional datasource as an OSGi service with the interface **javax.sql.DataSource**.

7. To build the **derby-ds** bundle and install it in the local Maven repository, enter the following Maven command from the **derby-ds** directory:

```
mvn install
```

10.4. DEFINE A TRANSACTIONAL ROUTE

Overview

This section describes how to create a transactional route and package it as an OSGi bundle. The route described here is based on the **AccountService** class (see Appendix A), implementing a transfer of funds from one account to another, where the account data is stored in an Apache Derby database instance.

Database schema

The database schema for the accounts consists of just two columns: the **name** column (identifying the account holder) and the **amount** column (specifying the amount of money left in the account). Formally, the schema is defined by the following SQL command:

```
CREATE TABLE accounts (name VARCHAR(50), amount INT);
```

Sample incoming message

The following XML snippet demonstrates the format of a typical message that is processed by the route:

```
<transaction>
  <transfer>
    <sender>Major Clanger</sender>
    <receiver>Tiny Clanger</receiver>
```

```
    <amount>90</amount>
  </transfer>
</transaction>
```

The message requests a transfer of money from one account to another. It specifies that 90 units should be subtracted from the **Major Clanger** account and 90 units should be added to the **Tiny Clanger** account.

The transactional route

The incoming messages are processed by the following transactional route:

```
<route>
  <from uri="jmstx:queue:giro">
  <bean ref="accountService" method="credit">
  <bean ref="accountService" method="debit">
  <bean ref="accountService" method="dumpTable">
  <to uri="jmstx:queue:statusLog">
</route>
```

Money is transferred by calling the **AccountService.credit** and **AccountService.debit** bean methods (which access the Derby database). The **AccountService.dumpTable** method then dumps the complete contents of the database table into the current exchange and the route sends this to the **statusLog** queue.

Provoking a transaction rollback

The **AccountService.debit** method imposes a limit of 100 on the amount that can be withdrawn from any account and throws an exception if this limit is exceeded. This provides a simple means of provoking a transaction rollback, by sending a message containing a transfer request that exceeds 100.

Steps to define a transactional route

Perform the following steps to define a route that uses XA to coordinate global transactions across a JMS XA resource and an Apache Derby XA resource:

1. Use the quickstart archetype to create a basic Maven project for the route bundle. Open a new command prompt, change directory to a convenient location, and enter the following command:

   ```
   mvn archetype:generate
     -DarchetypeArtifactId=maven-archetype-quickstart
     -DgroupId=org.fusesource.example
     -DartifactId=tx-xa
   ```

   The preceding command creates a new Maven project for the **org.fusesource.example/tx-xa** artifact under the **tx-xa** directory.

2. Change the project packaging type to **bundle**. Under the **tx-xa** directory, open the **pom.xml** file with a text editor and change the contents of the **packaging** element from **jar** to **bundle**, as shown in the following highlighted line:

   ```
   <project ...>
     ...
     <groupId>org.fusesource.example</groupId>
   ```

```
  <artifactId>tx-xa</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>bundle</packaging>
  ...
</project>
```

3. Add the bundle configuration to the POM. In the **pom.xml** file, add the following **build** element as a child of the **project** element:

```
<project ...>
  ...
  <build>
    <defaultGoal>install</defaultGoal>

    <plugins>

      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>${project.groupId}.${project.artifactId}</Bundle-
SymbolicName>
            <Import-Package>
              org.springframework.core,
              org.apache.camel,
              org.apache.camel.component.jms,
              org.apache.activemq,
              org.apache.activemq.xbean,
              org.apache.activemq.pool,
              org.apache.xbean.spring,
              org.apache.commons.pool,
              *
            </Import-Package>
            <Private-Package>
              org.fusesource.example.*
            </Private-Package>
            <DynamicImport-Package>
              org.apache.activemq.*
            </DynamicImport-Package>
          </instructions>
        </configuration>
      </plugin>

    </plugins>
  </build>

</project>
```

4. Customize the Maven compiler plug-in to enforce JDK 1.7 coding syntax. In the **pom.xml** file, add the following **plugin** element as a child of the **plugins** element, to configure the Maven compiler plug-in:

```
<project ...>
```

```
  ...
  <build>
    <defaultGoal>install</defaultGoal>

    <plugins>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>

</project>
```

5. Add the JBoss Fuse Bill of Materials (BOM) as the parent POM. The JBoss Fuse BOM defines version properties (for example, **camel-version**, **spring-version**, and so on) for all of the JBoss Fuse components, which makes it easy to specify the correct versions for the Maven dependencies. Add the following **parent** element near the top of your POM and (if necessary) customize the version of the BOM:

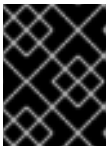
```
<project ...>
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.jboss.fuse.bom</groupId>
    <artifactId>jboss-fuse-parent</artifactId>
    <version>6.3.0.redhat-187</version>
  </parent>
  ...
</project>
```

6. Add the required Maven dependencies to the POM and specify the **derby-version** property. In the **pom.xml** file, add the following elements as children of the **project** element:

```
<project ...>
  ...
  <name>Global transactions demo</name>
  <url>redhat.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <derby-version>10.10.1.1</derby-version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
```

```
    <scope>test</scope>
  </dependency>

 <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-core</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-blueprint</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>${slf4j-version}</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>${log4j-version}</version>
  </dependency>

  <!-- Spring transaction dependencies -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring-version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring-version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${spring-version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>${spring-version}</version>
  </dependency>

  <!-- Spring JDBC adapter -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring-version}</version>
  </dependency>

  <!-- Database dependencies -->
  <dependency>
    <groupId>org.apache.derby</groupId>
```

```
    <artifactId>derby</artifactId>
    <version>${derby-version}</version>
  </dependency>

  <!-- JMS/ActiveMQ artifacts -->
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jms</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-camel</artifactId>
    <version>${activemq-version}</version>
  </dependency>
  <!--  This is needed by the camel-jms component -->
  <dependency>
    <groupId>org.apache.xbean</groupId>
    <artifactId>xbean-spring</artifactId>
    <version>${xbean-version}</version>
  </dependency>
 </dependencies>
 ...
</project>
```



IMPORTANT

Remember to customize the **derby-version** property to the version of Derby you are using.

7. Define the **AccountService** class. Under the **tx-xa** project directory, create the following directory:

   ```
   src/main/java/org/fusesource/example/tx/xa
   ```

   Create the file, **AccountService.java**, in this directory and add the contents of the listing from Example B.1, “The AccountService Class” to this file.

8. Define the beans and resources needed by the route in a Blueprint XML file. Under the **tx-xa** project directory, create the following directory:

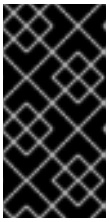
   ```
   src/main/resources/OSGI-INF/blueprint
   ```

   Using a text editor, create the file, **beans.xml**, in this directory and add the following contents to the file:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

     <!--
       JMS non-TX endpoint configuration
     -->
     <bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
   ```

```
        <property name="configuration" ref="jmsConfig">
    </bean>

    <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
        <property name="connectionFactory" ref="jmsPoolConnectionFactory">
    </bean>

    <!-- connection factory wrapper to support pooling -->
    <bean id="jmsPoolConnectionFactory"
class="org.apache.activemq.pool.PooledConnectionFactory">
        <property name="connectionFactory" ref="jmsConnectionFactory">
    </bean>

    <bean id="jmsConnectionFactory"
class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="vm:amq">
        <property name="userName" value="UserName">
        <property name="password" value="Password">
    </bean>


    <!--
        OSGi TM Service
    -->
    <!-- access through Spring's PlatformTransactionManager -->
    <reference id="osgiPlatformTransactionManager"
            interface="org.springframework.transaction.PlatformTransactionManager">
    <!-- access through javax TransactionManager -->
    <reference id="recoverableTxManager"

interface="org.apache.geronimo.transaction.manager.RecoverableTransactionManager">

    <!--
        JMS TX endpoint configuration
    -->
    <bean id="jmstx" class="org.apache.activemq.camel.component.ActiveMQComponent">
        <property name="configuration" ref="jmsTxConfig">
    </bean>

    <bean id="jmsTxConfig" class="org.apache.camel.component.jms.JmsConfiguration">
        <property name="connectionFactory" ref="jmsXaPoolConnectionFactory">
        <property name="transactionManager" ref="osgiPlatformTransactionManager">
        <property name="transacted" value="false">
        <property name="cacheLevelName" value="CACHE_CONNECTION">
    </bean>

    <!-- connection factory wrapper to support auto-enlisting of XA resource -->
    <bean id="jmsXaPoolConnectionFactory"
class="org.apache.activemq.pool.JcaPooledConnectionFactory">
        <!-- Defines the name of the broker XA resource for the Aries txn manager -->
        <property name="name" value="amq-broker">
        <property name="maxConnections" value="1">
        <property name="connectionFactory" ref="jmsXaConnectionFactory">
        <property name="transactionManager" ref="recoverableTxManager">
    </bean>
```

```
    <bean id="jmsXaConnectionFactory"
class="org.apache.activemq.ActiveMQXAConnectionFactory">
        <property name="brokerURL" value="vm:amq">
        <property name="userName" value="UserName">
        <property name="password" value="Password">
        <property name="redeliveryPolicy">
          <bean class="org.apache.activemq.RedeliveryPolicy">
            <property name="maximumRedeliveries" value="0">
          </bean>
        </property>
    </bean>

    <!--
      ActiveMQ XA Resource Manager
    -->
    <bean id="resourceManager"
class="org.apache.activemq.pool.ActiveMQResourceManager" init-
method="recoverResource">
        <property name="transactionManager" ref="recoverableTxManager">
        <property name="connectionFactory" ref="jmsXaConnectionFactory">
        <property name="resourceName" value="activemq.default">
    </bean>

    <!--
      Import Derby data sources as OSGi services
    -->
    <reference id="derbyXADataSource"
            interface="javax.sql.DataSource"
            filter="(datasource.name=derbyXADB)">
    <reference id="derbyDataSource"
            interface="javax.sql.DataSource"
            filter="(datasource.name=derbyDB)">


    <!-- bean for account business logic -->
    <bean id="accountService" class="org.fusesource.example.tx.xa.AccountService">
      <property name="dataSource" ref="derbyXADataSource">
    </bean>

</blueprint>
```



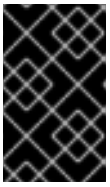
IMPORTANT

In the **jmsConnectionFactory** bean and in the **jmsXaConnectionFactory** bean, you must customize the JAAS user credentials, ***UserName*** and ***Password***, that are used to log into the broker. You can use any JAAS user with the **admin** role (usually defined in the **etc/users.properties** file of your JBoss Fuse installation).

9. Define the transactional route. In the **src/main/resources/OSGI-INF/blueprint** directory, create the new file, **camelContext.xml**, and add the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:camel="http://camel.apache.org/schema/blueprint">
```

```
  <camelContext xmlns="http://camel.apache.org/schema/blueprint" trace="false">
    <!-- Transactional route -->
    <route>
      <from uri="jmstx:queue:giro">
      <bean ref="accountService" method="credit">
      <bean ref="accountService" method="debit">
      <bean ref="accountService" method="dumpTable">
      <to uri="jmstx:queue:statusLog">
    </route>

    <!-- Feeder route -->
    <route>
      <from uri="file:PathNameToMsgDir">
      <to uri="jms:queue:giro">
    </route>
  </camelContext>

</blueprint>
```



IMPORTANT

Replace *PathNameToMsgDir* with the absolute path name of a temporary directory. When the application is running, you will use this directory as a convenient way of feeding XML messages into the route.

10. To build the **tx-xa** bundle and install it in the local Maven repository, enter the following Maven command from the **tx-xa** directory:

```
mvn install
```

10.5. DEPLOY AND RUN THE TRANSACTIONAL ROUTE

Overview

After creating the Derby database instance, you are ready to deploy the OSGi bundles into the container and test the route, as described here.

Steps to deploy and run the transactional route

Perform the following steps to deploy and run the transactional route in the Red Hat JBoss Fuse OSGi container:

1. Create the Derby database instance for the tutorial and create the **accounts** table, as follows:
    1. Open a command prompt and change directory to the Derby system directory that you specified earlier (that is, the value of the **derby.system.home** system property).
    2. Start the Derby database client utility, **ij**, by entering the following command:

```
ij
```

NOTE

By default, **ij** takes the current working directory to be the Derby system directory.

3. Create the **txXaTutorial** database instance, by entering the following **ij** command:

```
ij> CONNECT 'jdbc:derby:txXaTutorial;create=true';
```

4. Create the **accounts** table and create two sample row entries, by entering the following sequence of **ij** commands:

```
ij> CREATE TABLE accounts (name VARCHAR(50), amount INT);

ij> INSERT INTO accounts (name,amount) VALUES ('Major Clanger',2000);

ij> INSERT INTO accounts (name,amount) VALUES ('Tiny Clanger',100);
```

5. Exit **ij**, by entering the following command (don't forget the semicolon):

```
ij> EXIT;
```

2. Open a new command prompt and start the JBoss Fuse OSGi container by entering the following command:

```
./fuse
```

3. Install the **transaction** feature into the OSGi container. Enter the following console command:

```
JBossFuse:karaf@root> features:install transaction
```

4. Install the **connector** feature, which provides automatic XA datasource enlisting (Aries datasource wrapper). Enter the following console command:

```
JBossFuse:karaf@root> features:install connector
```

5. Install the **spring-jdbc** feature into the OSGi container. Enter the following console command:

```
JBossFuse:karaf@root> features:install spring-jdbc
```

6. Install the **derby** bundle into the OSGi container. Enter the following console command, replacing the bundle version with whatever version of Derby you are using:

```
JBossFuse:karaf@root> install mvn:org.apache.derby/derby/10.10.1.1
```

7. Install and start the **derby-ds** bundle (assuming that you have already built the bundle, as described in Section 10.3, “Define a Derby Datasource”) by entering the following console command:

```
JBossFuse:karaf@root> install -s mvn:org.fusesource.example/derby-ds/1.0-SNAPSHOT
```

8. To check that the datasources have been successfully exported from the **derby-ds** bundle, list the **derby-ds** services using the **osgi:ls** command. For example, given that *BundleID* is the bundle ID for the **derby-ds** bundle, you would enter the following console command:

   ```
   JBossFuse:karaf@root> osgi:ls BundleID
   ```

   Amongst the exported services, you should see an entry like the following:

   ```
   ----
   aries.managed = true
   aries.xa.aware = true
   aries.xa.name = derbyDS
   datasource.name = derbyXADB
   objectClass = [javax.sql.DataSource]
   osgi.service.blueprint.compname = derbyXADataSource
   service.id = 609
   service.ranking = 1000
   ----
   ```

   This is the wrapped XA datasource (recognizable from the **aries.xa.aware = true** setting), which is automatically created by the Aries wrapper feature (see Apache Aries Auto-Enlisting XA Wrapper).

9. Install and start the **tx-xa** bundle, by entering the following console command:

   ```
   JBossFuse:karaf@root> install -s mvn:org.fusesource.example/tx-xa
   ```

10. Create a file called **giro1.xml** in any convenient directory and use a text editor to add the following message contents to it:

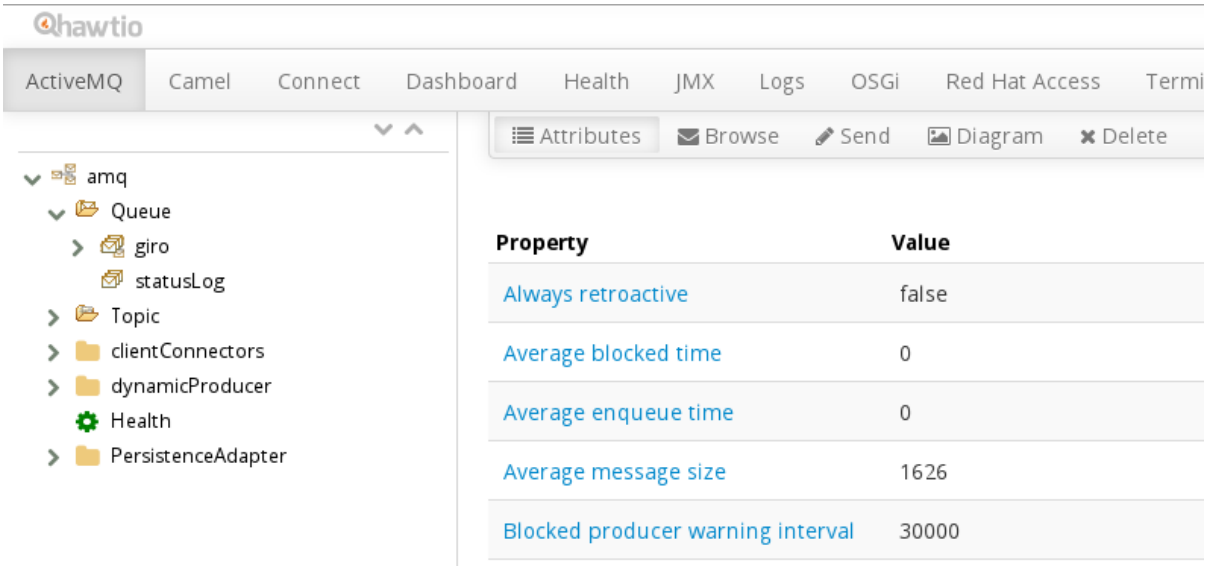
    ```
    <transaction>
      <transfer>
        <sender>Major Clanger</sender>
        <receiver>Tiny Clanger</receiver>
        <amount>90</amount>
      </transfer>
    </transaction>
    ```

    Now copy **giro1.xml** into the *PathNameToMsgDir* directory you created earlier (see Section 10.4, “Define a Transactional Route”). The **giro1.xml** file should disappear immediately after it is copied, because the *PathNameToMsgDir* is being monitored by the feeder route.

11. Use the Fuse Management Console to see what has happened to the message from **giro1.xml**. User your browser to navigate to the following URL: http://localhost:8181/hawtio. Login to the console using any valid JAAS username/password credentials (which are normally defined in the **etc/users.properties** file).

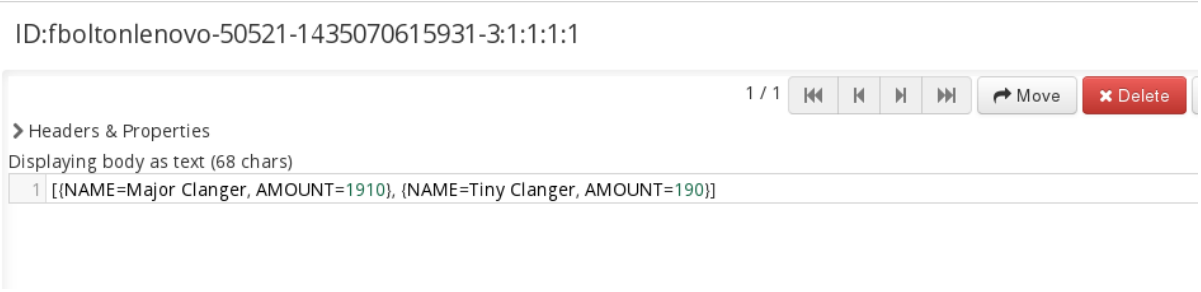
12. On the main menu bar, click on the **ActiveMQ** tab. In the left-hand pane of this view, drill down to the **statusLog** queue, as shown.

Figure 10.1. View of the statusLog Queue in Hawtio



13. On the menu bar above the right-hand pane, click **Browse** to browse the messages in the **statusLog** queue and click on the first message to view its contents. The body contains the most recent result of calling the **AccountService.dumpTable()** method (which is called in the last step of the transactional route).

Figure 10.2. Browsing Message Contents in the statusLog Queue



14. You can also force a transaction rollback by sending a message that exceeds the **AccountService.debit()** limit (withdrawal limit) of 100. For example, create the file **giro2.xml** and add the following message contents to it:

```
<transaction>
  <transfer>
    <sender>Major Clanger</sender>
    <receiver>Tiny Clanger</receiver>
    <amount>150</amount>
  </transfer>
</transaction>
```

When you copy this file into the *PathNameToMsgDir* directory, the message never propagates through to the **statusLog** queue, because the transaction gets rolled back.

APPENDIX A. OPTIMIZING PERFORMANCE OF JMS SINGLE- AND MULTIPLE-RESOURCE TRANSACTIONS

Abstract

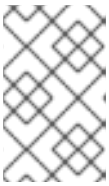
There are many ways you can optimize the performance of JMS transactions—both single-source transactions that use the resource's internal transaction manager and multiple-resource transactions that use an external XA transaction manager.

OPTIMIZATION TIPS FOR ALL JMS TRANSACTIONS

These tips apply to both single- and multiple-resource transactions:

- When working with Spring JMS/**camel-jms**, use a pooling-enabled Connection Factory, such as ActiveMQ's **PooledConnectionFactory**, to prevent clients from reopening JMS connections to the broker for each message consumed.
- When using **camel-jms** to do local transactions through an external transaction manager, configure the transaction manager to use a pooling-enabled Connection Factory.
- ActiveMQ's **PooledConnectionFactory** eagerly fills the internal connection pool, the size of which is bound by the maxConnections property setting.

  Each call to **PooledConnectionFactory.createConnection()** creates and adds a new connection to the pool until maxConnections is reached, regardless of the number of connections in use. Once maxConnections is reached, the pool hands out only connections it recycles. So different JMS sessions (in different threads) can potentially share the same JMS connection, which the JMS specification specifically allows.

  NOTE

  The **PooledConnectionFactory** also pools JMS sessions, but the session pool behaves differently than the connection pool. The session pool creates new sessions only when there are no free sessions in it.

- A **camel-jms** consumer configuration needs only one JMS connection from the connection pool, regardless of the setting of the concurrentConsumers property. As multiple camel routes can share the same **PooledConnectionFactory**, you can configure the pool's maxConnections property equal to the number of Camel routes sharing it.
- For local (transacted="true") JMS transaction using the **activemq** component, duplicate messages can occur on failover from master to slave. To prevent this from happening, set the lazyCreateTransactionManager property to **false**. **NOTE**: Do not set this property to **false** for XA transactions

OPTIMIZATION TIPS FOR JMS XA TRANSACTIONS

These tips apply to multiple-resource transactions only:

- With XA transactions, you must use CACHE_NONE or CACHE_CONNECTION in **camel-jms** or plain Spring JMS configurations. As CACHE_CONSUMER is not supported in these configurations, you need to use a pooling-enabled JMS Connection Factory to avoid opening a

new connection for every message consumed.

- Configure a **prefetch** of **1** on the ActiveMQ ConnectionFactory when not caching JMS consumers to eliminate any overhead generated by eager dispatch of messages to consumers. For example,

```
failover:(tcp://localhost:61616)?jms.prefetchPolicy.all=1
```

- When working with JMS providers other than ActiveMQ, wrap the 3rd-party JMS drivers in the generic XA-aware JcaPooledConnectionFactory (for details, see Section 6.6, “Generic XA-Aware Connection Pool Library”). For example, to wrap a WebSphereMQ endpoint:

```
<bean id="FuseWmqXaConnectionFactory"
class="org.apache.activemq.jms.pool.JcaPooledConnectionFactory">
  <property name="connectionFactory" ref="WMQConnectionFactory"/>
  <property name="transactionManager" ref="transactionManager"/>
  <property name="maxConnections" value="5"/>
  <!-- note we set a unique name for the XA resource" -->
  <property name="name" value="ibm-wmq" />
</bean>
<bean id="WMQConnectionFactory" class="com.ibm.mq.jms.MQXAConnectionFactory">
  <property name="hostName" value="localhost" />
  <property name="port" value="1414" />
  <property name="queueManager" value="QM_VM" />
  <property name="channel" value="TEST" />
  <property name="transportType" value="1" />
</bean>
```

  The wrapper provides the means for the Aries/Geronimo transaction manager to access and write the name of the participating WebSphereMQ resource to its HOWL recovery log file. This and other information stored in its recovery log file enables Aries to recover any pending transactions after a crash occurs.

- Always declare a resource manager for each resource involved in an XA transaction. Without a resource manager, pending XA transactions cannot be recovered after a JBoss Fuse crash, resulting in lost messages.

APPENDIX B. ACCOUNTSERVICE EXAMPLE

Abstract

The **AccountService** example class illustrates how you can use Spring **JdbcTemplate** class to access a JDBC data source.

B.1. ACCOUNTSERVICE EXAMPLE CODE

Overview

The **AccountService** class provides a simple example of accessing a data source through JDBC. The methods in this class can be used inside a local transaction or inside a global (XA) transaction.

Database schema

The **AccountService** example requires a single database table, **accounts**, which has two columns: a **name** column (containing the account name), and an **amount** column (containing the dollar balance of the account). The required database schema can be created by the following SQL statement:

```
CREATE TABLE accounts (name VARCHAR(50), amount INT);
```

AccountService class

Example B.1, “The AccountService Class” shows the complete listing of the **AccountService** class, which uses the Spring **JdbcTemplate** class to access a JDBC data source.

Example B.1. The AccountService Class

```
// Java
package org.fusesource.example.tx.xa;

import java.util.List;

import javax.sql.DataSource;


import org.apache.camel.Exchange;
import org.apache.camel.language.XPath;
import org.apache.log4j.Logger;
import org.springframework.jdbc.core.JdbcTemplate;


public class AccountService {
    private static Logger log = Logger.getLogger(AccountService.class);
    private JdbcTemplate jdbc;

    public AccountService() {
    }

    public void setDataSource(DataSource ds) {
        jdbc = new JdbcTemplate(ds);
```

```
    }

    public void credit(
            @XPath("/transaction/transfer/receiver/text()") String name,
            @XPath("/transaction/transfer/amount/text()") String amount
            )
    {
        log.info("credit() called with args name = " + name + " and amount = " + amount);
        int origAmount = jdbc.queryForInt(
                "select amount from accounts where name = ?",
                new Object[]{name}
        );
        int newAmount = origAmount + Integer.parseInt(amount);

        jdbc.update(
                "update accounts set amount = ? where name = ?",
                new Object[] {newAmount, name}
        );
    }

    public void debit(
            @XPath("/transaction/transfer/sender/text()") String name,
            @XPath("/transaction/transfer/amount/text()") String amount
            )
    {
        log.info("debit() called with args name = " + name + " and amount = " + amount);
        int iamount = Integer.parseInt(amount);
        if (iamount > 100) {
            throw new IllegalArgumentException("Debit limit is 100");
        }
        int origAmount = jdbc.queryForInt(
                "select amount from accounts where name = ?",
                new Object[]{name}
        );
        int newAmount = origAmount - Integer.parseInt(amount);
        if (newAmount < 0) {
            throw new IllegalArgumentException("Not enough in account");
        }

        jdbc.update(
                "update accounts set amount = ? where name = ?",
                new Object[] {newAmount, name}
        );
    }

    public void dumpTable(Exchange ex) {
        log.info("dump() called");
        List<?> dump = jdbc.queryForList("select * from accounts");
        ex.getIn().setBody(dump.toString());
    }
}
```

INDEX

C

J