HOW-TO

Distributed transactions in Spring, with and without XA

Seven transaction-processing patterns for Spring applications

By Dr. David Syer

JavaWorld | Jan 6, 2009 12:00 AM PT

A shared database resource can sometimes be synthesized from existing separate resources, especially if they are all in the same RDBMS platform. Enterprise-level database vendors all support the notion of synonyms (or the equivalent), where tables in one schema (to use the Oracle terminology) are declared as synonyms in another. In that way data that is partitioned physically in the platform can be addressed transactionally from the same `Connection` in a JDBC client. For example, the implementation of the shared-resource pattern with ActiveMQ in a real system (as opposed to the sample) would usually involve creating synonyms for the messaging and business data.

Performance and the JDBCPersistenceAdapter

Some people in the ActiveMQ community claim that the `JDBCPersistenceAdapter` creates performance problems. However, many projects and live systems use ActiveMQ with a relational database. In these cases the

received wisdom is that a journaled version of the adapter should be used to improve performance. This is not amenable to the shared-resource pattern (because the journal itself is a new transactional resource). However, the jury may still be out on the `JDBCPersistenceAdapter.` And indeed there are reasons to think that the use of a shared resource might *improve* performance over the journaled case. This is an area of active research among the Spring and ActiveMQ engineering teams.

Another shared-resource technique in a nonmessaging scenario (multiple databases) is to use the Oracle database link feature to link two database schemas together at the level of the RDBMS platform (see Resources). This may require changes to application code, or the creation of synonyms, because the table name aliases that refer to a linked database include the name of the link.

Best Efforts 1PC pattern

The Best Efforts 1PC pattern is fairly general but can fail in some circumstances that the developer must be aware of. This is a non-XA pattern that involves a synchronized single-phase commit of a number of resources. Because the 2PC is not used, it can never be as safe as an XA transaction, but is often good enough if the participants are aware of the compromises. Many high-volume, high-throughput transaction-processing systems are set up this way to improve performance.

The basic idea is to delay the commit of all resources as late as possible in a transaction so that the only thing that can go wrong is an infrastructure failure (not a business-processing error). Systems that rely on Best Efforts 1PC reason that infrastructure failures are rare enough that they can afford to take the risk in return for higher throughput. If business-processing services are also designed to be idempotent, then little can go wrong in practice.

To help you understand the pattern better and analyze the consequences of failure, I'll use the message-driven database update as an example.

The two resources in this transaction are counted in and counted out. The message transaction is started before the database one, and they end (either commit or rollback) in reverse order. So the sequence in the case of success might be the same as the one in at the beginning of this article:

1. Start messaging transaction
2. **Receive message**
3. Start database transaction
4. **Update database**
5. Commit database transaction
6. Commit messaging transaction

Actually, the order of the first four steps isn't crucial, except that the message must be received before the database is updated, and each transaction must start before its corresponding resource is used. So this sequence is just as valid:

1. Start messaging transaction
2. Start database transaction

3. **Receive message**
4. **Update database**
5. Commit database transaction
6. Commit messaging transaction

The point is just that the last two steps are important: they must come last, in this order. The reason why the ordering is important is technical, but the order itself is determined by business requirements. The order tells you that one of the transactional resources in this case is special; it contains instructions about how to carry out the work on the other. This is a business ordering: the system cannot tell automatically which way round it is (although if messaging and database are the two resources then it is often in this order). The reason the ordering is important has to do with the failure cases. The most common failure case (by far) is a failure of the business processing (bad data, programming error, and so on). In this case both transactions can easily be rigged to respond to an exception and rollback. In that case the integrity of the business data is preserved, with the timeline being similar to the ideal failure case outlined at the beginning of this article.

The precise mechanism for triggering the rollback is unimportant; several are available. The important point is that the commit or rollback happens in the reverse order of the business ordering in the resources. In the sample application, the messaging transaction must commit last because the instructions for the business process are contained in that resource. This is important because of the (rare) failure case in which the first commit succeeds and the second one fails. Because by design all business processing has already completed at this point, the only cause for such a partial failure would be an infrastructural problem with the messaging middleware.

Note that if the commit of the database resource fails, then the net effect is still a rollback. So the only nonatomic failure mode is one where the first transaction commits and the second rolls back. More generally, if there are n resources in the transaction, there are n-1 such failure modes leaving some of the resources in an inconsistent (committed) state after a rollback. In the message-database use case, the result of this failure mode is that the message is rolled back and comes back in another transaction, even though it was already successfully processed. So you can safely assume that the worse thing that can happen is that duplicate messages can be delivered. In the more general case, because the earlier resources in the transaction are considered to be potentially carrying information about how to carry out processing on the later resources, the net result of the failure mode can generically be referred to as *duplicate message*.

Some people take the risk that duplicate messages will happen infrequently enough that they don't bother trying to anticipate them. To be more confident about the correctness and consistency of your business data, though, you need to be aware of them in the business logic. If the business processing is aware that duplicate messages might arrive, all it has to do (usually at some extra cost, but not as much as the 2PC) is check whether it has processed that data before and do nothing if it has. This specialization is sometimes referred to as the Idempotent Business Service pattern.

The sample codes includes two examples of synchronizing transactional resources using this pattern. I'll discuss each in turn and then examine some other options.

Spring and message-driven POJOs

In the sample code's `best-jms-db project`, the participants are set up using mainstream configuration options so that the Best Efforts 1PC pattern is followed. The idea is that messages sent to a queue are picked up by an asynchronous listener and used to insert data into a table in the database.

The `TransactionAwareConnectionFactoryProxy` -- a stock component in Spring designed to be used in this pattern -- is the key ingredient. Instead of using the raw vendor-provided `ConnectionFactory`, the configuration wraps `ConnectionFactory` in a decorator that handles the transaction synchronization. This happens in the `jms-context.xml`, as shown in Listing 6:

Listing 6. Configuring a `TransactionAwareConnectionFactoryProxy` to wrap a vendor-provided JMS `ConnectionFactory`

```
<bean id="connectionFactory"
  class="org.springframework.jms.connection.TransactionA
  <property name="targetConnectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionF
      <property name="brokerURL" value="vm://localhost"/
    </bean>
  </property>
  <property name="synchedLocalTransactionAllowed" value=
</bean>
```

There is no need for the `ConnectionFactory`to know which transaction manager to synchronize with, because only one transaction active will be active at the time that it is needed, and Spring can handle that internally. The driving transaction is handled by a normal `DataSourceTransactionManager` configured in `data-`

source-context.xml. The component that needs to be aware of the transaction manager is the JMS listener container that will poll and receive messages:

```
<jms:listener-container transaction-manager="transaction
  <jms:listener destination="async" ref="fooHandler" met
</jms:listener-container>
```

The fooHandler and method tell the listener container which method on which component to call when a message arrives on the "async" queue. The handler is implemented like this, accepting a String as the incoming message, and using it to insert a record:

```
public void handle(String msg) {

  jdbcTemplate.update(
      "INSERT INTO T_FOOS (ID, name, foo_date) values (?
```


To simulate failures, the code uses a FailureSimulator aspect. It checks the message content to see if it supposed to fail, and in what way. The maybeFail() method, shown in Listing 7, is called after the FooHandler handles the message, but before the transaction has ended, so that it can affect the transaction's outcome:

Listing 7. The maybeFail() method

```
@AfterReturning("execution(* *..*Handler+.handle(String)
public void maybeFail(String msg) {
  if (msg.contains("fail")) {
    if (msg.contains("partial")) {
      simulateMessageSystemFailure();
    } else {
      simulateBusinessProcessingFailure();
    }
  }
}
```

The `simulateBusinessProcessingFailure()` method just throws a `DataAccessException` as if the database access had failed. When this method is triggered, you expect a full rollback of all database and message transactions. This scenario is tested in the sample project's `AsynchronousMessageTriggerAndRollbackTests` unit test.

`eSystemFailure()` method simulates a failure em by crippling the underlying JMS `Session`. The expected outcome here is a partial commit: the database work stays committed but the messages roll back. This is tested in the `AsynchronousMessageTriggerAndPartialRollbackTests` unit test.

The sample package also includes a unit test for the successful commit of all transactional work, in the `AsynchronousMessageTriggerSunnyDayTests` class.

The same JMS configuration and the same business logic can also be used in a synchronous setting, where the messages are received in a blocking call inside the business logic instead of delegating to a listener container. This approach is also demonstrated in the `best-`

`jms-db` sample project. The sunny-day case and the full rollback are tested in `SynchronousMessageTriggerSunnyDayTests` and `SynchronousMessageTriggerAndRollbackTests`, respectively.

Chaining transaction managers

In the other sample of the Best Efforts 1PC pattern (the `best-db-db` project) a crude implementation of a transaction manager just links together a list of other transaction managers to implement the transaction synchronization. If the business processing is successful they all commit, and if not they all roll back.

The implementation is in `ChainedTransactionManager`, which accepts a list of other transaction managers as an injected property, is shown in Listing 8:

Listing 8. Configuration of the ChainedTransactionManager

```
<bean id="transactionManager" class="com.springsource.op
  <property name="transactionManagers">
    <list>
      <bean
        class="org.springframework.jdbc.datasource.DataS
        <property name="dataSource" ref="dataSource" />
      </bean>
      <bean
        class="org.springframework.jdbc.datasource.DataS
        <property name="dataSource" ref="otherDataSource
      </bean>
    </list>
  </property>
</bean>
```

The simplest test for this configuration is just to insert something in both databases, roll back, and check that both operations left no trace. This is implemented as a unit test in `MulipleDataSourceTests`, the same as in the XA sample's `atomikos-db` project. The test fails if the rollback was not synchronized but the commit happened to work out.

Remember that the order of the resources is significant. They are nested, and the commit or rollback happens in reverse order to the order they are enlisted (which is the order in the configuration). This makes one of the resources special: the outermost resource always rolls back if there is a problem, even if the only problem is a failure of that resource. Also, the `testInsertWithCheckForDuplicates()` test method shows an idempotent business process protecting the system from partial failures. It is implemented as a defensive check in the business operation on the inner resource (the `otherDataSource` in this case):

```
int count = otherJdbcTemplate.update("UPDATE T_AUDITS ..
if (count == 0) {
  count = otherJdbcTemplate.update("INSERT into T_AUDITS
}
```

The update is tried first with a `where` clause. If nothing happens, the data you hoped to find in the update is inserted. The cost of the extra protection from the idempotent process in this case is one extra query (the update) in the sunny-day case. This cost would be quite low in a more complicated business process in which many queries are executed per transaction.