HOW-TO

Distributed transactions in Spring, with and without XA

Seven transaction-processing patterns for Spring applications

10

By Dr. David Syer

JavaWorld | Jan 6, 2009 12:00 AM PT

While it's common to use the Java Transaction API and the XA protocol for distributed transactions in Spring, you do have other options. The optimum implementation depends on the types of resources your application uses and the trade-offs you're willing to make between performance, safety, reliability, and data integrity. In this JavaWorld feature, SpringSource's David Syer guides you through seven patterns for distributed transactions in Spring applications, three of them with XA and four without. *Level: Intermediate*

The Spring Framework's support for the Java Transaction API (JTA) enables applications to use distributed transactions and the XA protocol without running in a Java EE container. Even with this support, however, XA is expensive and can be unreliable or cumbersome to administrate. It may come as a welcome surprise, then, that a certain class of applications can avoid the use of XA altogether.

To help you understand the considerations involved in various approaches to distributed transactions, I'll analyze seven transaction-processing patterns, providing code samples to make them concrete. I'll present the patterns in reverse order of safety or reliability, starting with those with the highest guarantee of data integrity and atomicity under the most general circumstances. As you move down the list, more caveats and limitations will apply. The patterns are also roughly in reverse order of runtime cost (starting with the most expensive). The patterns are all architectural, or technical, as opposed to business patterns, so I don't focus on the business use case, only on the minimal amount of code to see each pattern working.

Note that only the first three patterns involve XA, and those might not be available or acceptable on performance grounds. I don't discuss the XA patterns as extensively as the others because they are covered elsewhere, though I do provide a simple demonstration of the first one. By reading this article you'll learn what you can and can't do with distributed transactions and how and when to avoid the use of XA -- and when not to.

Distributed transactions and atomicity

A *distributed transaction* is one that involves more than one transactional resource. Examples of transactional resources are the connectors for communicating with relational databases and messaging middleware. Often such a resource has an API that looks something like `begin()`, `rollback()`, `commit()`. In the Java world, a transactional resource usually shows up as the product of a factory provided by the underlying platform: for a database, it's a `Connection` (produced by `DataSource`) or Java Persistence API (JPA) `EntityManager`; for Java Message Service (JMS), it's a `Session`.

In a typical example, a JMS message triggers a database update. Broken down into a timeline, a successful interaction goes something like this:

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

If a database error such as a constraint violation occurred on the update, the desirable sequence would look like this:

1. Start messaging transaction
2. **Receive message**
3. Start database transaction
4. **Update database, fail!**
5. Roll back database transaction
6. Roll back messaging transaction

In this case, the message goes back to the middleware after the last rollback and returns at some point to be received in another transaction. This is usually a good thing, because otherwise you might have no record that a failure occurred. (Mechanisms to deal with automatic retry and handling exceptions are out of this article's scope.)

The important feature of both timelines is that they are *atomic*, forming a single logical transaction that either succeeds completely or fails completely.

But what guarantees that the timeline looks like either of these sequences? Some synchronization between the transactional resources must occur, so that if one commits they both do, and vice versa. Otherwise, the whole transaction is not atomic. The transaction is distributed because multiple resources are involved, and without synchronization it will not be atomic. The technical and conceptual difficulties with distributed transactions all relate to the synchronization of the resources (or lack of it).

The first three patterns discussed below are based on the XA protocol. Because these patterns have been widely covered, I won't go into much detail about them here. Those familiar with XA patterns may want to skip ahead to the Shared Transaction Resource pattern.

Full XA with 2PC

If you need close-to-bulletproof guarantees that your application's transactions will recover after an outage, including a server crash, then Full XA is your only choice. The shared resource that is used to synchronize the transaction in this case is a special transaction manager that coordinates information about the process using the XA protocol. In Java, from the developer's point of view, the protocol is exposed through a JTA `UserTransaction`.

Being a system interface, XA is an enabling technology that most developers never see. They need to know is that it's there, what it enables, what it costs, and the implications for how they use transactional resources. The cost comes from the two-phase commit (2PC) protocol that the transaction manager uses to ensure that all resources agree on the outcome of a transaction before it ends.

If the application is Spring-enabled, it uses the Spring `JtaTransactionManager` and Spring declarative transaction management to hide the details of the underlying synchronization. The difference for the developer between using XA and not using XA is all about configuring the factory resources: the `DataSource` instances, and the transaction manager for the application. This article includes a sample application (the `atomikos-db` project) that illustrates this configuration. The `DataSource` instances and the transaction manager are the only XA- or JTA-specific elements of the application.

To see the sample working, run the unit tests under `com.springsource.open.db`. A simple `MulipleDataSourceTests` class just inserts data into two data sources and then uses the Spring integration support features to roll back the transaction, as shown in Listing 1:

Listing 1. Transaction rollback

```
@Transactional
  @Test
  public void testInsertIntoTwoDataSources() throws Exception {

    int count = getJdbcTemplate().update(
        "INSERT into T_FOOS (id,name,foo_date) values (?,?,null)", 0,
        "foo");
    assertEquals(1, count);

    count = getOtherJdbcTemplate()
        .update(
            "INSERT into T_AUDITS (id,operation,name,audit_date) values (
            0, "INSERT", "foo", new Date());
    assertEquals(1, count);

    // Changes will roll back after this method exits

  }
```

Then `MulipleDataSourceTests` verifies that the two operations were both rolled back, as shown in Listing 2:

Listing 2. Verifying rollback

```
@AfterTransaction
  public void checkPostConditions() {

    int count = getJdbcTemplate().queryForInt("select count(*) from T_FOO
    // This change was rolled back by the test framework
    assertEquals(0, count);

    count = getOtherJdbcTemplate().queryForInt("select count(*) from T_AU
    // This rolled back as well because of the XA
    assertEquals(0, count);

  }
```

For a better understanding of how Spring transaction management works and how to configure it generally, see the Spring Reference Guide.

XA with 1PC Optimization

This pattern is an optimization that many transaction managers use to avoid the overhead of 2PC if the transaction includes a single resource. You would expect your application server to be able to figure this out.

XA and the Last Resource Gambit

Another feature of many XA transaction managers is that they can still provide the same recovery guarantees when all but one resource is XA-capable as they can when they all are. They do this by ordering the resources and using the non-XA resource as a casting vote. If it fails to commit, then all the other resources can be rolled back. It is close to 100 percent bulletproof -- but is not quite that. And when it fails, it fails without leaving much of a trace unless extra steps are taken (as is done in some of the top-end implementations).

Shared Transaction Resource pattern

A great pattern for decreasing complexity and increasing throughput in some systems is to remove the need for XA altogether by ensuring that all the transactional resources in the system are actually backed by the same resource. This is clearly not possible in all processing use cases, but it is just as solid as XA and usually much faster. The Shared Transaction Resource pattern is bulletproof but specific to certain platforms and processing scenarios.

A simple and familiar (to many) example of this pattern is the sharing of a database `Connection` between a component that uses object-relational mapping (ORM) with a component that uses JDBC. This is what happens you use the Spring transaction managers that support the ORM tools such as Hibernate, EclipseLink, and the Java Persistence API (JPA). The same transaction can safely be used across ORM and JDBC components, usually driven from above by a service-level method execution where the transaction is controlled.

Another effective use of this pattern is the case of message-driven update of a single database (as in the simple example in this article's introduction). Messaging-middleware systems need to store their data somewhere, often in a relational database. To implement this pattern, all that's needed is to point the messaging system at the same database the business data is going into. This pattern relies on the messaging-middleware vendor exposing the details of its storage strategy so that it can be configured to point to the same database and hook into the same transaction.

Not all vendors make this easy. An alternative, which works for almost any database, is to use Apache ActiveMQ for messaging and plug a storage strategy into the message broker. This is fairly easy to configure once you know the trick. It's demonstrated in this article's `shared-jms-db` samples project. The application code (unit tests in this case) does not need to be aware that this pattern is in use, because it is all enabled declaratively in Spring configuration.

A unit test in the sample called `SynchronousMessageTriggerAndRollbackTests` verifies that everything is working with synchronous message reception. The `testReceiveMessageUpdateDatabase` method receives two messages and uses them to insert two records in the database. When this method exits, the test framework rolls back the transaction, so you can verify that the messages and the database updates are both rolled back, as shown in Listing 3:

Listing 3. Verifying rollback of messages and database updates

```
@AfterTransaction
public void checkPostConditions() {

  assertEquals(0, SimpleJdbcTestUtils.countRowsInTable(jdbcTemplate, "T_F
  List<String> list = getMessages();
  assertEquals(2, list.size());

}
```

The most important features of the configuration are the ActiveMQ persistence strategy, linking the messaging system to the same `DataSource` as the business data, and the flag on the Spring `JmsTemplate` used to receive the messages. Listing 4 shows how to configure the ActiveMQ persistence strategy:

Listing 4. Configuring ActiveMQ persistence

```
<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectic
  depends-on="brokerService">
  <property name="brokerURL" value="vm://localhost?async=false" />
</bean>

<bean id="brokerService" class="org.apache.activemq.broker.BrokerService"
  destroy-method="stop">
    ...
  <property name="persistenceAdapter">
    <bean class="org.apache.activemq.store.jdbc.JDBCPersistenceAdapter">
      <property name="dataSource">
        <bean class="com.springsource.open.jms.JmsTransactionAwareDataSou
          <property name="targetDataSource" ref="dataSource"/>
          <property name="jmsTemplate" ref="jmsTemplate"/>
        </bean>
      </property>
      <property name="createTablesOnStartup" value="true" />
    </bean>
  </property>
</bean>
```

Listing 5 shows the flag on the Spring `JmsTemplate` that is used to receive the messages.

Listing 5. Configuring the `JmsTemplate` for transactional use

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  ...
  <!-- This is important... -->
  <property name="sessionTransacted" value="true" />
</bean>
```

Without `sessionTransacted=true`, the JMS session transaction API calls will never be made and the message reception cannot be rolled back. The important ingredients here are the embedded broker with a special `async=false` parameter and a wrapper for the `DataSource` that together ensure that ActiveMQ uses the same transactional JDBC `Connection` as Spring.

View 10 Comments