JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-01-2009/jw-01-spring-transactions.html

Distributed transactions in Spring, with and without XA

Seven transaction-processing patterns for Spring applications

By Dr. David Syer, JavaWorld.com, 01/06/09

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_FOOS");
    // This change was rolled back by the test framework
    assertEquals(0, count);

    count = getOtherJdbcTemplate().queryForInt("select count(*) from T_AUDITS");
    // 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_FOOS"));
  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.ActiveMQConnectionFactory"
  depends-on="brokerService">
  <property name="brokerURL" value="vm://localhost?async=false" />
</bean>

<bean id="brokerService" class="org.apache.activemq.broker.BrokerService" init-method="start"
  destroy-method="stop">
    ...
  <property name="persistenceAdapter">
    <bean class="org.apache.activemq.store.jdbc.JDBCPersistenceAdapter">
      <property name="dataSource">
        <bean class="com.springsource.open.jms.JmsTransactionAwareDataSourceProxy">
          <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. Setting up 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.

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.TransactionAwareConnectionFactoryProxy">
  <property name="targetConnectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory" depends-on="brokerService">
      <property name="brokerURL" value="vm://localhost"/>
    </bean>
  </property>
  <property name="synchedLocalTransactionAllowed" value="true" />
</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="transactionManager" >
  <jms:listener destination="async" ref="fooHandler" method="handle"/>
</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 (?, ?,?)", count.getAndIncrement(), msg, new Date());

}
```

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)) && args(msg)")
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.

The `simulateMessageSystemFailure()` method simulates a failure in the messaging system 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.open.db.ChainedTransactionManager">
  <property name="transactionManagers">
    <list>
      <bean
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
      </bean>
      <bean
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <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 ... WHERE id=, ...?");
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.

Other options

The `ChainedTransactionManager` in the sample has the virtue of simplicity; it doesn't bother with many extensions and optimizations that are available. An alternative approach is to use the `TransactionSychronization` API in Spring to register a callback for the current transaction when the second resource joins. This is the approach in the `best-jms-db` sample, where the key feature is the combination of `TransactionAwareConnectionFactory` with a `DataSourceTransactionManager`. This special case

could be expanded on and generalized to include non-JMS resources using the `TransactionSynchronizationManager`. The advantage would be that in principle only those resources that joined the transaction would be enlisted, instead of all resources in the chain. However, the configuration would still need to be aware of which participants in a potential transaction correspond to which resources.

Also, the Spring engineering team is considering a "Best Efforts 1PC transaction manager" feature for the Spring Core. You can vote for the JIRA issue if you like the pattern and want to see explicit and more transparent support for it in Spring.

Nontransactional Access pattern

The Nontransactional Access pattern needs a special kind of business process in order to make sense. The idea is that sometimes one of the resources that you need to access is marginal and doesn't need to be in the transaction at all. For instance, you might need to insert a row into an audit table that's independent of whether the business transaction is successful or not; it just records the attempt to do something. More commonly, people overestimate how much they need to make read-write changes to one of the resources, and quite often read-only access is fine. Or else the write operations can be carefully controlled, so that if anything goes wrong it can be accounted for or ignored.

In these cases the resource that stays outside the transaction probably actually has its own transaction, but it is not synchronized with anything else that is happening. If you are using Spring, the main transaction is driven by a `PlatformTransactionManager`, and the marginal resource might be a database `Connection` obtained from a `DataSource` not controlled by the transaction manager. All that happens is that each access to the marginal resource has the default setting of `autoCommit=true`. Read operations won't see updates that are happening concurrently in another uncommitted transaction (assuming reasonable default isolation levels), but the effect of write operations will normally be seen immediately by other participants.

This pattern requires more careful analysis, and more confidence in designing the business processes, but it isn't all that different from the Best Efforts 1PC. A generic service that provides compensating transactions when anything goes wrong is too ambitious a goal for most projects. But simple use cases involving services that are idempotent and execute only one write operation (and possibly many reads) are not that uncommon. These are the ideal situations for a nontransactional gambit.

Wing-and-a-Prayer: An antipattern

The last pattern is really an antipattern. It tends to occur when developers don't understand distributed transactions or don't realize that they have one. Without an explicit call to the underlying resource's transaction API, you can't just assume that all the resources will join a transaction. If you are using a Spring transaction manager other than `JtaTransactionManager`, it will have one transactional resource attached to it. That transaction manager will be the one that is used to intercept method executions using Spring declarative transaction management features like `@Transactional`. No other resources can be expected to be enlisted in the same transaction. The usual outcome is that everything works just fine on a sunny day, but as soon as there is an exception the user finds that one of the resources didn't roll back. A typical mistake leading to this problem is using a `DataSourceTransactionManager` and a repository implemented with Hibernate.

Which pattern to use?

I'll conclude by analyzing the pros and cons of the patterns introduced, to help you see how to decide between them. The first step is to recognize that you have a system requiring distributed transactions. A necessary (but not sufficient) condition is that there is a single process with more than one transactional resource. A sufficient condition is that those resources are used together in a single use case, normally driven by a call into the service level in your architecture.

If you haven't recognized the distributed transaction, you have probably implemented the **Wing-and-a-Prayer** pattern. Sooner or later you will see data that should have been rolled back but wasn't. Probably when you see the effect it will be a long way downstream from the actual failure, and quite hard to trace back. The Wing-and-a-Prayer can also be inadvertently used by developers who believe they are protected by XA but haven't configured the underlying resources to participate in the transaction. I worked on a project once where the database had been installed by another group, and they had switched off the XA support in the installation process. Everything ran just fine for a few months and then strange failures started to creep into the business process. It took a long time to diagnose the problem.

If your use cases with mixed resources are simple enough and you can afford to do the analysis and perhaps some refactoring, then the **Nontransactional Resource** pattern might be an option. This works best when one of the resources is read-mostly, and the write operations can be guarded with checks for duplicates. The data in the nontransactional resource must make sense in business terms even after a failure. Audit, versioning, and logging information typically fits into this category. Failures will be relatively common (any time anything in the real transaction rolls back), but you can be confident that there are no side effects.

Best Efforts 1PC is for systems that need more protection from common failures but don't want the overhead of 2PC. Performance improvements can be significant. It is more tricky to set up than a Nontransactional Resource, but it shouldn't require as much analysis and is used for more generic data types. Complete certainty about data consistency requires that business processing is idempotent for "outer" resources (any but the first to commit). Message-driven database updates are a perfect example and have quite good support already in Spring. More unusual scenarios require some additional framework code (which may eventually be part of Spring).

The **Shared Resource** pattern is perfect for special cases, normally involving two resources of a particular type and platform (such as. ActiveMQ with any RDBMS or Oracle AQ co-located with an Oracle database). The benefits are extreme robustness and excellent performance.

Sample code updates

The sample code provided with this article will inevitably show its age as new versions of Spring and other components are released. See the Spring Community Site to access the author's up-to-date code, as well as current versions of the Spring Framework and related components.

Full XA with 2PC is generic and will always give the highest confidence and greatest protection against failures where multiple, diverse resources are being used. The downside is that it is expensive because of additional I/O prescribed by the protocol (but don't write it off until you try it) and requires special-purpose platforms. There are open source JTA implementations that can provide a way to break free of the application server, but many developers consider them second best, still. It is certainly the case that more people use JTA and XA than need to if they could spend more time thinking about the transaction boundaries in their systems. At least if they use Spring their business logic doesn't need to be aware of how the transactions are handled, so platform choices can be deferred.

About the author

Dr. David Syer is a Principal Consultant with SpringSource, based in the UK. He is a founder and lead engineer on the Spring Batch project, an open source framework for building and configuring offline and batch-processing applications. He is a frequent presenter at conferences on Enterprise Java and commentator on the industry. Recent publications appeared in The Server Side, InfoQ and the SpringSource blog.