



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

Page 3 of 3

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 `TransactionSynchronization` 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.

 *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*

JAWORLD

*g and configuring offline and batch-processing
... frequent presenter at conferences on Enterprise Java and
commentator on the industry. Recent publications appeared in The Server Side,
InfoQ and the SpringSource blog.*

Learn more about this topic

- Download the source code
(<http://images.techhive.com/downloads/idge/imported/article/jvw/2009/01/springxa-src.zip>) for this article. Also be sure to visit the Spring Community Site (<http://www.springframework.org/>) for the most up-to-date sample code for this article.
- Learn more about `javax.transaction` from the Java docs for JTA (<http://java.sun.com/javaee/5/docs/api/javax/transaction/package-frame.html>) and XAResource (<http://java.sun.com/javase/6/docs/api/javax/transaction/xa/XAResource.html>).
- "XA transactions using Spring (<http://www.javaworld.com/javaworld/jw-04-2007/jw-04-xa.html>)" (Murali Kosaraju, JavaWorld, April 2007) explains how to set up Spring with JTA outside a Java EE container.
- "XA Exposed, Part I (<http://iroller.com/pyrasun/category/XA>)" (Mike Spille, Pyrasun, The Spille Blog, April 2004) is an excellent and entertaining resource for learning about 2PC in more detail.
- Learn more about how Spring transaction management works and how to configure it generally by reading the *Spring Reference Guide*, Chapter 9. Transaction management (<http://static.springframework.org/spring/docs/2.5.x/reference/transaction.html>).
- "Transaction management for J2EE 1.2 (<http://www.javaworld.com/jw-07-2000/jw-0714-transaction.html>)" (Sanjay Mahapatra, JavaWorld, July 2000) defines the ACID properties of a transaction, including atomicity.
- In "To XA or not to XA (<http://guysblogspot.blogspot.com/2006/10/to-xa-or-not-to-xa.html>)" (Guy's Blog, October 2006), Atomikos CTO Guy Pardon advocates for using XA.

- Check out the [Atomikos documentation](http://www.atomikos.com/Documentation/WebHome) (<http://www.atomikos.com/Documentation/WebHome>) to learn about this open source transaction manager.
- ["How to create a database link in Oracle"](http://searchoracle.techtarget.com/tip/0,289483,sid41_gci1263933,00.html) (http://searchoracle.techtarget.com/tip/0,289483,sid41_gci1263933,00.html) (Elisa Gabbert. SearchOracle.com, January 2004) explains how to create an Oracle



[Sign In](#) | [Register](#)

JAWORLD

vide a "best efforts" 1PC transaction manager out of the box (<http://jira.springframework.org/browse/SPR-3844>) proposal for the Spring Framework.

More from JavaWorld

- See the [JavaWorld Site Map](http://www.javaworld.com/topics/) (<http://www.javaworld.com/topics/>) for a complete listing of research centers focused on client-side, enterprise, and core Java development tools and topics.
- [JavaWorld's Java Technology Insider](http://www.javaworld.com/podcasts/jtech/) (<http://www.javaworld.com/podcasts/jtech/>) is a podcast series that lets you learn from Java technology experts on your way to work.
- [Network World's IT Product Guides](http://www.networkworld.com/productguides/index.html) (<http://www.networkworld.com/productguides/index.html>) offer side-by-side comparison of hundreds of products in over 50 categories.

Follow everything from JavaWorld



[View Comments](#)