IBM

developerWorks®

# Transaction strategies: Understanding transaction pitfalls

## Beware of these common mistakes when implementing transactions in the Java platform

Mark Richards

February 03, 2009

Transaction processing should achieve a high degree of data integrity and consistency. This article, the first in a series on developing an effective transaction strategy for the Java platform, introduces common transaction pitfalls that can prevent you from reaching this goal. Using code examples from the Spring Framework and the Enterprise JavaBeans (EJB) 3.0 specification, series author Mark Richards explains these all-too-common mistakes.

View more content in this series

The most common reason for using transactions in an application is to maintain a high degree of data integrity and consistency. If you're unconcerned about the quality of your data, you needn't concern yourself with transactions. After all, transaction support in the Java platform can kill performance, introduce locking issues and database concurrency problems, and add complexity to your application.

But developers who don't concern themselves with transactions do so at their own peril. Almost all business-related applications require a high degree of data quality. The financial investment industry alone wastes tens of billions of dollars on failed trades, with bad data being the second-leading cause. Although lack of transaction support is only one factor leading to bad data (albeit a major one), a safe inference is that billions of dollars are wasted in the financial investment industry alone as a result of nonexistent or poor transaction support.

**About this series**

Transactions improve the quality, integrity, and consistency of your data and make your applications more robust. Implementation of successful transaction processing in Java applications is not a trivial exercise, and it's about design as much as about coding. In this new series, Mark Richards is your guide to designing an effective transaction strategy for use cases ranging from simple applications to high-performance transaction processing.

Ignorance about transaction support is another source of problems. All too often I hear claims like "we don't need transaction support in our applications because they never fail." Right. I have witnessed some applications that in fact rarely or never throw exceptions. These applications bank on well-written code, well-written validation routines, and full testing and code coverage support to avoid the performance costs and complexity associated with transaction processing. The problem with this type of thinking is that it takes into account only one characteristic of transaction support: *atomicity*. Atomicity ensures that all updates are treated as a single unit and are either all committed or all rolled back. But rolling back or coordinating updates isn't the only aspect of transaction support. Another aspect, *isolation*, ensures that one unit of work is isolated from other units of work. Without proper transaction isolation, other units of work can access updates made by an ongoing unit of work, even though that unit of work is incomplete. As a result, business decisions might be made on the basis of partial data, which could cause failed trades or other negative (or costly) outcomes.

**Better late than never**

I started to appreciate the problems with transaction processing in early 2000 when, while working at a client site, I noticed a line item on the project plan right above the system testing task. It read *implement transaction support*. Sure, easy enough to add transaction support to a major application when it is almost ready for system testing, right? Unfortunately, this approach is all too common. At least this project, unlike most, *was* implementing transaction support, albeit at the end of the development cycle.

So, given the high cost and negative impact of bad data and the basic knowledge that transactions are important (and necessary), you need to use transactions and learn how to deal with the issues that can arise. You press on and add transaction support to your applications. And that's where the problem often begins. Transactions don't always seem to work as promised in the Java platform. This article is an exploration of the reasons why. With the help of code examples, I'll introduce some of the common transaction pitfalls I continually see and experience in the field, in most cases in production environments.

Although most of this article's code examples use the Spring Framework (version 2.5), the transaction concepts are the same as for the EJB 3.0 specification. In most cases, it is simply a matter of replacing the Spring Framework `@Transactional` annotation with the `@TransactionAttribute` annotation found in the EJB 3.0 specification. Where the two frameworks differ in concept and technique, I have included both Spring Framework and EJB 3.0 source code examples.

## Local transaction pitfalls

A good place to start is with the easiest scenario: the use of *local transactions*, also commonly referred to as *database transactions*. In the early days of database persistence (for example, JDBC), we commonly delegated transaction processing to the database. After all, isn't that what the database is supposed to do? Local transactions work fine for logical units of work (LUW) that perform a single insert, update, or delete statement. For example, consider the simple JDBC code in Listing 1, which performs an insert of a stock-trade order to a `TRADE` table:

## Listing 1. Simple database insert using JDBC

```
@Stateless
public class TradingServiceImpl implements TradingService {
   @Resource SessionContext ctx;
   @Resource(mappedName="java:jdbc/tradingDS") DataSource ds;

   public long insertTrade(TradeData trade) throws Exception {
      Connection dbConnection = ds.getConnection();
      try {
         Statement sql = dbConnection.createStatement();
         String stmt =
            "INSERT INTO TRADE (ACCT_ID, SIDE, SYMBOL, SHARES, PRICE, STATE)"
          + "VALUES ("
          + trade.getAcct() + "','"
          + trade.getAction() + "','"
          + trade.getSymbol() + "',"
          + trade.getShares() + ","
          + trade.getPrice() + ",'"
          + trade.getState() + "')";
         sql.executeUpdate(stmt, Statement.RETURN_GENERATED_KEYS);
         ResultSet rs = sql.getGeneratedKeys();
         if (rs.next()) {
            return rs.getBigDecimal(1).longValue();
         } else {
            throw new Exception("Trade Order Insert Failed");
         }
      } finally {
         if (dbConnection != null) dbConnection.close();
      }
   }
}
```

The JDBC code in Listing 1 includes no transaction logic, yet it persists the trade order in the `TRADE` table in the database. In this case, the database handles the transaction logic.

This is all well and good for a single database maintenance action in the LUW. But suppose you need to update the account balance at the same time you insert the trade order into the database, as shown in Listing 2:

## Listing 2. Performing multiple table updates in the same method

```
public TradeData placeTrade(TradeData trade) throws Exception {
   try {
      insertTrade(trade);
      updateAcct(trade);
      return trade;
   } catch (Exception up) {
      //log the error
      throw up;
   }
}
```

In this case, the `insertTrade()` and `updateAcct()` methods use standard JDBC code without transactions. Once the `insertTrade()` method ends, the database has persisted (and committed) the trade order. If the `updateAcct()` method should fail for any reason, the trade order would remain in the `TRADE` table at the end of the `placeTrade()` method, resulting in inconsistent data in the database. If the `placeTrade()` method had used transactions, both of these activities would have been included in a single LUW, and the trade order would have been rolled back if the account update failed.

With the popularity of Java persistence frameworks like Hibernate, TopLink, and the Java Persistence API (JPA) on the rise, we rarely write straight JDBC code anymore. More commonly, we use the newer object-relational mapping (ORM) frameworks to make our lives easier by replacing all of that nasty JDBC code with a few simple method calls. For example, to insert the trade order from the JDBC code example in Listing 1, using the Spring Framework with JPA, you'd map the `TradeData` object to the `TRADE` table and replace all of that JDBC code with the JPA code in Listing 3:

### Listing 3. Simple insert using JPA

```
public class TradingServiceImpl {
    @PersistenceContext(unitName="trading") EntityManager em;

    public long insertTrade(TradeData trade) throws Exception {
       em.persist(trade);
       return trade.getTradeId();
    }
}
```

Notice that Listing 3 invokes the `persist()` method on the `EntityManager` to insert the trade order. Simple, right? Not really. This code will not insert the trade order into the `TRADE` table as expected, nor will it throw an exception. It will simply return a value of `0` as the key to the trade order without changing the database. This is one of the first major pitfalls of transaction processing: *ORM-based frameworks require a transaction in order to trigger the synchronization between the object cache and the database*. It is through a transaction commit that the SQL code is generated and the database affected by the desired action (that is, insert, update, delete). Without a transaction there is no trigger for the ORM to generate SQL code and persist the changes, so the method simply ends — no exceptions, no updates. If you are using an ORM-based framework, you must use transactions. You can no longer rely on the database to manage the connections and commit the work.

These simple examples should make it clear that transactions are necessary in order to maintain data integrity and consistency. But they only begin to scratch the surface of the complexity and pitfalls associated with implementing transactions in the Java platform.

## Spring Framework `@Transactional` annotation pitfalls

So, you test the code in Listing 3 and discover that the `persist()` method didn't work without a transaction. As a result, you view a few links from a simple Internet search and find that with the Spring Framework, you need to use the `@Transactional` annotation. So you add the annotation to your code as shown in Listing 4:

### Listing 4. Using the `@Transactional` annotation

```
public class TradingServiceImpl {
   @PersistenceContext(unitName="trading") EntityManager em;

   @Transactional
   public long insertTrade(TradeData trade) throws Exception {
      em.persist(trade);
      return trade.getTradeId();
   }
}
```

You retest your code, and you find it still doesn't work. The problem is that you must tell the Spring Framework that you are using annotations for your transaction management. Unless you are doing full unit testing, this pitfall is sometimes hard to discover. It usually leads to developers simply adding the transaction logic in the Spring configuration files rather than through annotations.

When using the `@Transactional` annotation in Spring, you must add the following line to your Spring configuration file:

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

The `transaction-manager` property holds a reference to the transaction manager bean defined in the Spring configuration file. This code tells Spring to use the `@Transaction` annotation when applying the transaction interceptor. Without it, the `@Transactional` annotation is ignored, resulting in no transaction being used in your code.

Getting the basic `@Transactional` annotation to work in the code in Listing 4 is only the beginning. Notice that Listing 4 uses the `@Transactional` annotation without specifying any additional annotation parameters. I've found that many developers use the `@Transactional` annotation without taking the time to understand fully what it does. For example, when using the `@Transactional` annotation by itself as I do in Listing 4, what is the transaction propagation mode set to? What is the read-only flag set to? What is the transaction isolation level set to? More important, when should the transaction roll back the work? Understanding how this annotation is used is important to ensuring that you have the proper level of transaction support in your application. To answer the questions I've just asked: when using the `@Transactional` annotation by itself without any parameters, the propagation mode is set to `REQUIRED`, the read-only flag is set to `false`, the transaction isolation level is set to the database default (usually `READ_COMMITTED`), and the transaction will not roll back on a checked exception.

## `@Transactional` read-only flag pitfalls

A common pitfall I frequently come across in my travels is the improper use of the read-only flag on the Spring `@Transactional` annotation. Here is a quick quiz for you: When using standard JDBC code for Java persistence, what does the `@Transactional` annotation in Listing 5 do when the read-only flag is set to `true` and the propagation mode set to `SUPPORTS`?

### Listing 5. Using read-only with `SUPPORTS` propagation mode — JDBC

```
@Transactional(readOnly = true, propagation=Propagation.SUPPORTS)
public long insertTrade(TradeData trade) throws Exception {
   //JDBC Code...
}
```

When the `insertTrade()` method in Listing 5 executes, does it:

- Throw a read-only connection exception
- Correctly insert the trade order and commit the data
- Do nothing because the propagation level is set to `SUPPORTS`

Give up? The correct answer is B. The trade order is correctly inserted into the database, even though the read-only flag is set to `true` and the transaction propagation set to `SUPPORTS`. But how can that be? No transaction is started because of the `SUPPORTS` propagation mode, so the method effectively uses a local (database) transaction. The read-only flag is applied only if a transaction is started. In this case, no transaction was started, so the read-only flag is ignored.

Okay, so if that is the case, what does the `@Transactional` annotation do in Listing 6 when the read-only flag is set and the propagation mode is set to `REQUIRED`?

### Listing 6. Using read-only with `REQUIRED` propagation mode — JDBC

```
@Transactional(readOnly = true, propagation=Propagation.REQUIRED)
public long insertTrade(TradeData trade) throws Exception {
   //JDBC code...
}
```

When executed, does the `insertTrade()` method in Listing 6:

- Throw a read-only connection exception
- Correctly insert the trade order and commit the data
- Do nothing because the read-only flag is set to `true`

This one should be easy to answer given the prior explanation. The correct answer here is A. An exception will be thrown, indicating that you are trying to perform an update operation on a read-only connection. Because a transaction is started (`REQUIRED`), the connection is set to read-only. Sure enough, when you try to execute the SQL statement, you get an exception telling you that the connection is a read-only connection.

The odd thing about the read-only flag is that you need to start a transaction in order to use it. Why would you need a transaction if you are only reading data? The answer is that you don't. Starting a transaction to perform a read-only operation adds to the overhead of the processing thread and can cause shared read locks on the database (depending on what type of database you are using and what the isolation level is set to). The bottom line is that the read-only flag is somewhat meaningless when you use it for JDBC-based Java persistence and causes additional overhead when an unnecessary transaction is started.

What about when you use an ORM-based framework? In keeping with the quiz format, can you guess what the result of the `@Transactional` annotation in Listing 7 would be if the `insertTrade()` method were invoked using JPA with Hibernate?

## Listing 7. Using read-only with `REQUIRED` propagation mode — JPA

```
@Transactional(readOnly = true, propagation=Propagation.REQUIRED)
public long insertTrade(TradeData trade) throws Exception {
   em.persist(trade);
   return trade.getTradeId();
}
```

Does the `insertTrade()` method in Listing 7:

- Throw a read-only connection exception
- Correctly insert the trade order and commit the data
- Do nothing because the `readOnly` flag is set to true

The answer to this question is a bit more tricky. In some cases the answer is C, but in most cases (particularly when using JPA) the answer is B. The trade order is correctly inserted into the database without error. Wait a minute — the preceding example shows that a read-only connection exception would be thrown when the `REQUIRED` propagation mode is used. That is true when you use JDBC. However, when you use an ORM-based framework, the read-only flag works a bit differently. When you are generating a key on an insert, the ORM framework will go to the database to obtain the key and subsequently perform the insert. For some vendors, such as Hibernate, the flush mode will be set to `MANUAL`, and no insert will occur for inserts with non-generated keys. The same holds true for updates. However, other vendors, like TopLink, will always perform inserts and updates when the read-only flag is set to true. Although this is both vendor and version specific, the point here is that you cannot be guaranteed that the insert or update will not occur when the read-only flag is set, particularly when using JPA as it is vendor-agnostic.

Which brings me to another major pitfall I frequently encounter. Given all you've read so far, what do you suppose the code in Listing 8 would do if you only set the read-only flag on the `@Transactional` annotation?

## Listing 8. Using read-only — JPA

```
@Transactional(readOnly = true)
public TradeData getTrade(long tradeId) throws Exception {
   return em.find(TradeData.class, tradeId);
}
```

Does the `getTrade()` method in Listing 8:

- Start a transaction, get the trade order, then commit the transaction
- Get the trade order without starting a transaction

**Never say never**

At certain times you may want to start a transaction for a database read operation — for example, when isolating your read operations for consistency or setting a specific transaction isolation level for the read operation. However, these situations are rare in business applications, and unless you're faced with one, you should avoid starting a transaction for database read operations, as they are unnecessary and can lead to database deadlocks, poor performance, and poor throughput.

The correct answer here is A. A transaction is started and committed. Don't forget: the default propagation mode for the `@Transactional` annotation is `REQUIRED`. This means that a transaction is started when in fact one is not required (see Never say never). . Depending on the database you are using, this can cause unnecessary shared locks, resulting in possible deadlock situations in the database. In addition, unnecessary processing time and resources are being consumed starting and stopping the transaction. The bottom line is that when you use an ORM-based framework, the read-only flag is quite useless and in most cases is ignored. But if you still insist on using it, always set the propagation mode to `SUPPORTS`, as shown in Listing 9, so no transaction is started:

### Listing 9. Using read-only and `SUPPORTS` propagation mode for select operation

```
@Transactional(readOnly = true, propagation=Propagation.SUPPORTS)
public TradeData getTrade(long tradeId) throws Exception {
   return em.find(TradeData.class, tradeId);
}
```

Better yet, just avoid using the `@Transactional` annotation altogether when doing read operations, as shown in Listing 10:

### Listing 10. Removing the `@Transactional` annotation for select operations

```
public TradeData getTrade(long tradeId) throws Exception {
   return em.find(TradeData.class, tradeId);
}
```

## `REQUIRES_NEW` transaction attribute pitfalls

Whether you're using the Spring Framework or EJB, use of the `REQUIRES_NEW` transaction attribute can have negative results and lead to corrupt and inconsistent data. The `REQUIRES_NEW` transaction attribute always starts a new transaction when the method is started, whether or not an existing transaction is present. Many developers use the `REQUIRES_NEW` attribute incorrectly, assuming it is the correct way to make sure that a transaction is started. Consider the two methods in Listing 11:

### Listing 11. Using the `REQUIRES_NEW` transaction attribute

```
@Transactional(propagation=Propagation.REQUIRES_NEW)
public long insertTrade(TradeData trade) throws Exception {...}

@Transactional(propagation=Propagation.REQUIRES_NEW)
public void updateAcct(TradeData trade) throws Exception {...}
```

Notice in Listing 11 that both of these methods are public, implying that they can be invoked independently from each other. Problems occur with the `REQUIRES_NEW` attribute when methods using it are invoked within the same logical unit of work via inter-service communication or through orchestration. For example, suppose in Listing 11 that you can invoke the `updateAcct()` method independently of any other method in some use cases, but there's also the case where the `updateAcct()` method is also invoked in the `insertTrade()` method. Now, if an exception occurs after the `updateAcct()` method call, the trade order would be rolled back, but the account updates would be committed to the database, as shown in Listing 12:

### Listing 12. Multiple updates using the `REQUIRES_NEW` transaction attribute

```
@Transactional(propagation=Propagation.REQUIRES_NEW)
public long insertTrade(TradeData trade) throws Exception {
   em.persist(trade);
   updateAcct(trade);
   //exception occurs here! Trade rolled back but account update is not!
   ...
}
```

This happens because a new transaction is started in the `updateAcct()` method, so that transaction commits once the `updateAcct()` method ends. When you use the `REQUIRES_NEW` transaction attribute, if an existing transaction context is present, the current transaction is suspended and a new transaction started. Once that method ends, the new transaction commits and the original transaction resumes.

Because of this behavior, the `REQUIRES_NEW` transaction attribute should be used only if the database action in the method being invoked needs to be saved to the database regardless of the outcome of the overlaying transaction. For example, suppose that every stock trade that was attempted had to be recorded in an audit database. This information needs to be persisted whether or not the trade failed because of validation errors, insufficient funds, or some other reason. If you did not use the `REQUIRES_NEW` attribute on the audit method, the audit record would be rolled back along with the attempted trade. Using the `REQUIRES_NEW` attribute guarantees that the audit data is saved regardless of the initial transaction's outcome. The main point here is always to use either the `MANDATORY` or `REQUIRED` attribute instead of `REQUIRES_NEW` unless you have a reason to use it for reasons similar those to the audit example.

## Transaction rollback pitfalls

I've saved the most common transaction pitfall for last. Unfortunately, I see this one in production code more times than not. I'll start with the Spring Framework and then move on to EJB 3.

So far, the code you have been looking at looks something like Listing 13:

### Listing 13. No rollback support

```
@Transactional(propagation=Propagation.REQUIRED)
public TradeData placeTrade(TradeData trade) throws Exception {
   try {
      insertTrade(trade);
      updateAcct(trade);
      return trade;
   } catch (Exception up) {
      //log the error
      throw up;
   }
}
```

Suppose the account does not have enough funds to purchase the stock in question or is not set up to purchase or sell stock yet and throws a checked exception (for example, `FundsNotAvailableException`). Does the trade order get persisted in the database or is the entire logical unit of work rolled back? The answer, surprisingly, is that upon a checked exception (either in the Spring Framework or EJB), the transaction commits any work that has not yet been committed. Using Listing 13, this means that if a checked exception occurs during the `updateAcct()` method, the trade order is persisted, but the account isn't updated to reflect the trade.

This is perhaps the primary data-integrity and consistency issue when transactions are used. Run-time exceptions (that is, unchecked exceptions) automatically force the entire logical unit of work to roll back, but checked exceptions do not. Therefore, the code in Listing 13 is useless from a transaction standpoint; although it appears that it uses transactions to maintain atomicity and consistency, in fact it does not.

Although this sort of behavior may seem strange, transactions behave this way for some good reasons. First of all, not all checked exceptions are bad; they might be used for event notification or to redirect processing based on certain conditions. But more to the point, the application code may be able to take corrective action on some types of checked exceptions, thereby allowing the transaction to complete. For example, consider the scenario in which you are writing the code for an online book retailer. To complete the book order, you need to send an e-mail confirmation as part of the order process. If the e-mail server is down, you would send some sort of SMTP checked exception indicating that the message cannot be sent. If checked exceptions caused an automatic rollback, the entire book order would be rolled back just because the e-mail server was down. By not automatically rolling back on checked exceptions, you can catch that exception and perform some sort of corrective action (such as sending the message to a pending queue) and commit the rest of the order.

When you use the Declarative transaction model (described in more detail in Part 2 of this series), you must specify how the container or framework should handle checked exceptions. In the Spring Framework you specify this through the `rollbackFor` parameter in the `@Transactional` annotation, as shown in Listing 14:

## Listing 14. Adding transaction rollback support — Spring

```
@Transactional(propagation=Propagation.REQUIRED, rollbackFor=Exception.class)
public TradeData placeTrade(TradeData trade) throws Exception {
   try {
      insertTrade(trade);
      updateAcct(trade);
      return trade;
   } catch (Exception up) {
      //log the error
      throw up;
   }
}
```

Notice the use of the `rollbackFor` parameter in the `@Transactional` annotation. This parameter accepts either a single exception class or an array of exception classes, or you can use the `rollbackForClassName` parameter to specify the names of the exceptions as Java `String` types. You can also use the negative version of this property (`noRollbackFor`) to specify that all exceptions should force a rollback except certain ones. Typically most developers specify `Exception.class` as the value, indicating that all exceptions in this method should force a rollback.

EJBs work a little bit differently from the Spring Framework with regard to rolling back a transaction. The `@TransactionAttribute` annotation found in the EJB 3.0 specification does not include directives to specify the rollback behavior. Rather, you must use the `SessionContext.setRollbackOnly()` method to mark the transaction for rollback, as illustrated in Listing 15:

### Listing 15. Adding transaction rollback support — EJB

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public TradeData placeTrade(TradeData trade) throws Exception {
   try {
      insertTrade(trade);
      updateAcct(trade);
      return trade;
   } catch (Exception up) {
      //log the error
      sessionCtx.setRollbackOnly();
      throw up;
   }
}
```

Once the `setRollbackOnly()` method is invoked, you cannot change your mind; the only possible outcome is to roll back the transaction upon completion of the method that started the transaction. The transaction strategies described in future articles in the series will provide guidance on when and where to use the rollback directives and on when to use the `REQUIRED` vs. `MANDATORY` transaction attributes.

## Conclusion

The code used to implement transactions in the Java platform is not overly complex; however, how you use and configure it can get somewhat complex. Many pitfalls are associated with implementing transaction support in the Java platform (including some less common ones that I haven't discussed here). The biggest issue with most of them is that no compiler warnings or run-time errors tell you that the transaction implementation is incorrect. Furthermore, contrary to the assumption reflected in the "Better late than never" anecdote at the start of this article, implementing transaction support is not only a coding exercise. A significant amount of design effort goes into developing an overall transaction strategy. The rest of the *Transaction strategies* series will help guide you in terms of how to design an effective transaction strategy for use cases ranging from simple applications to high-performance transaction processing.

## Related topic

- Spring Framework 2.5 documentation: Chapter 9. Transaction management