Transaction strategies: Models and strategies overview

Learn about the three transaction models and the transaction strategies that use them

Mark Richards

March 03, 2009

It's a common mistake to confuse transaction models with transaction strategies. This second article in the *Transaction strategies* series outlines the three transaction models supported by the Java™ platform and introduces four primary transaction strategies that use those models. Using examples from the Spring Framework and the Enterprise JavaBeans (EJB) 3.0 specification, Mark Richards explains how the transaction models work and how they can form the basis for developing transaction strategies ranging from basic transaction processing to high-speed transaction-processing systems.

View more content in this series

All too often, developers, designers, and architects confuse *transaction models* with *transaction strategies*. I typically ask the architect or technical lead in a client engagement to describe their project's transaction strategy. I usually get one of three responses. Sometimes it's a quiet "Oh, well, we really don't use transactions in our applications." Other times I hear a confused "Um, I'm really not sure what you mean." Usually, however, I get the confident response that "We are using declarative transactions." But as you will see in this article, the term *declarative transactions* describes a transaction *model*, but by no means is it a transaction *strategy*.

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.

The three transaction models supported by the Java platform are:

- The Local Transaction model

© Copyright IBM Corporation 2009

- The Programmatic Transaction model
- The Declarative Transaction model

These models describe the basics of how transactions should behave in the Java platform and how they are implemented. However, they provide only the rules and semantics for transaction processing. How the transaction model is applied is entirely up to you. For example, when should you use the `REQUIRED` vs. the `MANDATORY` transaction attribute? When and where should you specify the transaction rollback directives? When should you consider the Programmatic Transaction model vs. the Declarative Transaction model? How do you optimize transactions for high-performance systems? The transaction models themselves can't answer these questions. Rather, you must address them either by developing your own transaction strategy or by adopting one of the four primary transaction strategies I introduce in this article.

As you saw in the first article of this series, many common transaction pitfalls can affect transactional behavior and, consequently, diminish your data's integrity and consistency. Likewise, the lack of an effective (or any) transaction strategy will have a negative effect on your data's integrity and consistency. The transaction models I describe in this article are the building blocks for developing an effective transaction strategy. Understanding the differences among the models and how they work is paramount to understanding the transaction strategies that use them. After describing the three transaction models, I'll introduce four transaction strategies that apply to most business applications, ranging from simple Web applications to large high-speed transaction-processing systems. Subsequent articles in the *Transaction Strategies* series will describe these strategies in detail.

Local Transaction model

The Local Transaction model gets its name from the fact that transactions are managed by the underlying database resource manager, not the container or framework your application is running in. In this model, you manage *connections* rather than *transactions*. As you learned from "Understanding transaction pitfalls," you can't use the Local Transaction model when you make database updates using an object-relational mapping framework such as Hibernate, TopLink, or the Java Persistence API (JPA). You can still apply it when using data-access object (DAO) or JDBC-based frameworks and database stored procedures.

You can use the Local Transaction model in one of two ways: let the database manage the connection, or manage the connection programmatically. To let the database manage the connection, you set the `autoCommit` property on the JDBC `Connection` object to `true` (the default value), which tells the underlying database management system (DBMS) to commit the transaction after the insert, update, or delete has completed, or roll back the work if it fails. This technique is illustrated in Listing 1, which inserts a stock-trade order into a `TRADE` table:

Listing 1. Local transactions with a single update

```
public class TradingServiceImpl {
   public void processTrade(TradeData trade) throws Exception {
      Connection dbConnection = null;
      try {
         DataSource ds = (DataSource)
              (new InitialContext()).lookup("jdbc/MasterDS");
         dbConnection = ds.getConnection();
         dbConnection.setAutoCommit(true);
  Statement sql = dbConnection.createStatement();
  String stmt = "insert into TRADE ...";
  sql.executeUpdate(stmt1);
      } finally {
         if (dbConnection != null)
            dbConnection.close();
      }
   }
}
```

Notice in Listing 1 that the `autoCommit` value is set to `true`, indicating to the DBMS that the local transaction should be committed after each database statement. This technique works fine if you have a single database maintenance activity within the logical unit of work (LUW). However, suppose that the `processTrade()` method shown in Listing 1 also updates the balance in the `ACCT` table to reflect the trade order's value. In this case, the two database actions would be independent of each other, with the insert to the `TRADE` table being committed to the database before the update of the `ACCT` table. Should the update to the `ACCT` table fail, there would be no mechanism to roll back the insert to the `TRADE` table, resulting in inconsistent data in the database.

This scenario leads to the second technique: managing the connections programmatically. In this technique, you would set the `autoCommit` property on the `Connection` object to `false` and manually commit or roll back the connection. Listing 2 illustrates this technique:

Listing 2. Local transactions with multiple updates

```
public class TradingServiceImpl {
   public void processTrade(TradeData trade) throws Exception {
      Connection dbConnection = null;
      try {
         DataSource ds = (DataSource)
             (new InitialContext()).lookup("jdbc/MasterDS");
         dbConnection = ds.getConnection();
         dbConnection.setAutoCommit(false);
  Statement sql = dbConnection.createStatement();
  String stmt1 = "insert into TRADE ...";
  sql.executeUpdate(stmt1);
  String stmt2 = "update ACCT set balance...";
  sql.executeUpdate(stmt2);
  dbConnection.commit();
      } catch (Exception up) {
         dbConnection.rollback();
         throw up;
      } finally {
         if (dbConnection != null)
            dbConnection.close();
      }
   }
}
```

Notice in Listing 2 that the `autoCommit` property is set to `false`, informing the underlying DBMS that the connection will be managed in the code, not the database. In this case, you must invoke the `commit()` method on the `Connection` object if all is well; otherwise, invoke the `rollback()` method if an exception occurs. In this manner, you can coordinate the two database activities in the same unit of work.

Although the Local Transaction model may seem somewhat outdated in this day and age, it is an important element for one of the primary transaction strategies I'll introduce toward the end of the article.

Programmatic Transaction model

The Programmatic Transaction model gets its name from the fact that the developer is responsible for managing the transaction. In the Programmatic Transaction model, unlike the Local Transaction model, you manage *transactions* and are isolated from the underlying database *connections*.

Similar to the example in Listing 2, with this model the developer is responsible for obtaining a transaction from the transaction manager, starting the transaction, committing the transaction, and — if an exception occurs — rolling back the transaction. As you can probably guess, this results in a lot of error-prone code that tends to get in the way of the business logic in your applications. However, some transaction strategies require the use of the Programmatic Transaction model.

Although the concepts are the same, implementation of the Programmatic Transaction model differs between the Spring Framework and the EJB 3.0 specification. I'll illustrate the implementation of this model first using EJB 3.0, then show the same database updates using the Spring Framework.

Programmatic transactions with EJB 3.0

In EJB 3.0, you obtain a transaction from the transaction manager (in other words, the container) by doing a Java Naming and Directory Interface (JNDI) lookup on the `javax.transaction.UserTransaction`. Once you have a `UserTransaction`, you can invoke the `begin()` method to start the transaction, the `commit()` method to commit the transaction, and the `rollback()` method to roll back the transaction if an error occurs. In this model, the container will not automatically commit or roll back the transaction; it is up to the developer to program this behavior in the Java method performing the database updates. Listing 3 shows an example of the Programmatic Transaction model for EJB 3.0 using JPA:

Listing 3. Programmatic transactions using EJB 3.0

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class TradingServiceImpl implements TradingService {
   @PersistenceContext(unitName="trading") EntityManager em;

   public void processTrade(TradeData trade) throws Exception {
      InitialContext ctx = new InitialContext();
      UserTransaction txn = (UserTransaction)ctx.lookup("UserTransaction");
      try {
         txn.begin();
```

```
  em.persist(trade);
  AcctData acct = em.find(AcctData.class, trade.getAcctId());
  double tradeValue = trade.getPrice() * trade.getShares();
  double currentBalance = acct.getBalance();
  if (trade.getAction().equals("BUY")) {
     acct.setBalance(currentBalance - tradeValue);
         } else {
            acct.setBalance(currentBalance + tradeValue);
         }
         txn.commit();
      } catch (Exception up) {
         txn.rollback();
         throw up;
      }
   }
}
```

When using the Programmatic Transaction model in a Java Platform, Enterprise Edition (Java EE) container environment with a stateless session bean, you must tell the container you are using programmatic transactions. You do this by using the `@TransactionManagement` annotation and setting the transaction type to `BEAN`. If you don't use this annotation, the container assumes you are using declarative transaction management (`CONTAINER`), which is the default transaction type for EJB 3.0. When you use programmatic transactions in the client layer outside of the context of a stateless session bean, you don't need to set the transaction type.

Programmatic transactions with Spring

The Spring Framework has two ways of implementing the Programmatic Transaction model. One way is through the Spring `TransactionTemplate`, and the other is by using a Spring *platform transaction manager* directly. Because I am not a big fan of anonymous inner classes and hard-to-read code, I will use the second technique to illustrate the Programmatic Transaction model in Spring.

Spring has at least nine platform transaction managers. The most common ones you will most likely use are the `DataSourceTransactionManager`, `HibernateTransactionManager`, `JpaTransactionManager`, and the `JtaTransactionManager`. My code examples use JPA, so I'll show the configuration for the `JpaTransactionManager`.

To configure the `JpaTransactionManager` in Spring, simply define the bean in the application context XML file using the `org.springframework.orm.jpa.JpaTransactionManager` class and add a reference to the JPA Entity Manager Factory bean. Then, assuming the class containing your application logic is managed by Spring, inject the transaction manager into the bean, as shown in Listing 4:

Listing 4. Defining the Spring JPA transaction manager

```
<bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
   <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<bean id="tradingService" class="com.trading.service.TradingServiceImpl">
   <property name="txnManager" ref="transactionManager"/>
</bean>
```

If Spring doesn't manage the application class, you can obtain a reference to the transaction manager in your method by using the `getBean()` method on the Spring context.

In the source code, you can now use the platform manager to get a transaction. Once all the updates are performed you can invoke the `commit()` method to commit the transaction, or the `rollback()` method to roll back the transaction. Listing 5 illustrates this technique:

Listing 5. Using the Spring JPA transaction manager

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

   JpaTransactionManager txnManager = null;
   public void setTxnManager(JpaTransactionManager mgr) {
      txnManager = mgr;
   }

   public void processTrade(TradeData trade) throws Exception {
      TransactionStatus status =
         txnManager.getTransaction(new DefaultTransactionDefinition());
      try {
         em.persist(trade);
  AcctData acct = em.find(AcctData.class, trade.getAcctId());
  double tradeValue = trade.getPrice() * trade.getShares();
  double currentBalance = acct.getBalance();
  if (trade.getAction().equals("BUY")) {
     acct.setBalance(currentBalance - tradeValue);
  } else {
     acct.setBalance(currentBalance + tradeValue);
  }
         txnManager.commit(status);
      } catch (Exception up) {
         txnManager.rollback(status);
         throw up;
      }
   }
}
```

Notice in Listing 5 the difference between the Spring Framework and EJB 3.0. In Spring, the transaction is retrieved (and consequently started) by invoking the `getTransaction()` method on the platform transaction manager. The anonymous `DefaultTransactionDefinition` class contains details about the transaction and its behavior, including the transaction name, isolation level, propagation mode (transaction attribute), and transaction timeout (if any). In this case, I am simply using the default values, which are an empty string for the name, the default isolation level for the underlying DBMS (usually `READ_COMMITTED`), `PROPAGATION_REQUIRED` for the transaction attribute, and the default timeout of the DBMS. Also notice that the `commit()` and `rollback()` methods are invoked using the platform transaction manager, not the transaction (as is the case with EJB).

Declarative Transaction model

The Declarative Transaction model, otherwise known as *Container Managed Transactions* (CMT), is the most common transaction model in the Java platform. In this model, the container environment takes care of starting, committing, and rolling back the transaction. The developer is responsible only for specifying the transactions' behavior. Most of the transaction pitfalls discussed in this series' first article are associated with the Declarative Transaction model.

Both the Spring Framework and EJB 3.0 make use of annotations to specify the transaction behavior. Spring uses the `@Transactional` annotation, whereas EJB 3.0 uses the `@TransactionAttribute` annotation. The container will not automatically roll back a transaction on a checked exception when you use the Declarative Transaction model. The developer must specify where and when to roll back a transaction when a checked exception occurs. In the Spring Framework, you specify this by using the `rollbackFor` property on the `@Transactional` annotation. In EJB, you specify it by invoking the `setRollbackOnly()` method on the `SessionContext`.

Listing 6 illustrates the use of the Declarative Transaction model for EJB:

Listing 6. Declarative transactions using EJB 3.0

```
@Stateless
public class TradingServiceImpl implements TradingService {
   @PersistenceContext(unitName="trading") EntityManager em;
   @Resource SessionContext ctx;

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void processTrade(TradeData trade) throws Exception {
      try {
         em.persist(trade);
  AcctData acct = em.find(AcctData.class, trade.getAcctId());
  double tradeValue = trade.getPrice() * trade.getShares();
  double currentBalance = acct.getBalance();
  if (trade.getAction().equals("BUY")) {
     acct.setBalance(currentBalance - tradeValue);
  } else {
     acct.setBalance(currentBalance + tradeValue);
  }
      } catch (Exception up) {
         ctx.setRollbackOnly();
         throw up;
      }
   }
}
```

Listing 7 illustrates the use of the Declarative Transaction model for the Spring Framework:

Listing 7. Declarative transactions using Spring

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

   @Transactional(propagation=Propagation.REQUIRED,
                  rollbackFor=Exception.class)
   public void processTrade(TradeData trade) throws Exception {
      em.persist(trade);
      AcctData acct = em.find(AcctData.class, trade.getAcctId());
      double tradeValue = trade.getPrice() * trade.getShares();
      double currentBalance = acct.getBalance();
      if (trade.getAction().equals("BUY")) {
         acct.setBalance(currentBalance - tradeValue);
      } else {
         acct.setBalance(currentBalance + tradeValue);
      }
   }
}
```

Transaction attributes

In addition to the rollback directives, you must also specify the *transaction attribute*, which defines how the transaction should behave. The Java platform supports six types of transaction attributes, regardless of whether you are using EJB or the Spring Framework:

- `Required`
- `Mandatory`
- `RequiresNew`
- `Supports`
- `NotSupported`
- `Never`

In describing each of these transaction attributes, I'll use a fictitious method named `methodA()` that the transaction attribute is being applied to.

If the `Required` transaction attribute is specified for `methodA()` and `methodA()` is invoked under the scope of an existing transaction, the existing transaction scope will be used. Otherwise, `methodA()` will start a new transaction. If the transaction is started by `methodA()`, then it must also be terminated (committed or rolled back) by `methodA()`. This is the most commonly used transaction attribute and is the default for both EJB 3.0 and Spring. Unfortunately, in many cases, it is used incorrectly, resulting in data-integrity and consistency issues. For each of the transaction strategies I'll cover in subsequent articles in this series, I'll discuss use of this transaction attribute in more detail.

If the `Mandatory` transaction attribute is specified for `methodA()` and `methodA()` is invoked under an existing transaction's scope, the existing transaction scope will be used. However, if `methodA()` is invoked without a transaction context, then a `TransactionRequiredException` will be thrown, indicating that a transaction must be present before `methodA()` is invoked. This transaction attribute is used in the *Client Orchestration* transaction strategy described in this article's next section.

The `RequiresNew` transaction attribute is an interesting one. More often than not, I find this attribute misused or misunderstood. If the `RequiresNew` transaction attribute is specified for `methodA()` and `methodA()` is invoked with or without a transaction context, a new transaction will always be started (and terminated) by `methodA()`. This means that if `methodA()` is invoked within the context of another transaction (called `Transaction1` for example), `Transaction1` will be suspended and a new transaction (called `Transaction2`) will be started. Once `methodA()` ends, `Transaction2` is then either committed or rolled back, and `Transaction1` resumes. This clearly violates the ACID (atomicity, consistency, isolation, durability) properties of a transaction (specifically the *atomicity* property). In other words, all database updates are no longer contained within a single unit of work. If `Transaction1` were to be rolled back, the changes committed by `Transaction2` remain committed. If that's the case, what good is this transaction attribute? As indicated in the first article in this series, this transaction attribute should only be used for database operations (such as auditing or logging) that are independent of the underlying transaction (in this case `Transaction1`).

The `Supports` transaction attribute is another one that I find most developers don't fully understand or appreciate. If the `Supports` transaction attribute is specified for `methodA()` and `methodA()` is invoked within the scope of an existing transaction, `methodA()` will execute under the scope of that transaction. However, if `methodA()` is invoked without a transaction context, then no transaction will be started. This attribute is primarily used for read-only operations to the database. If that's the case, why not specify the `NotSupported` transaction attribute (described in the next paragraph) instead? After all, that attribute guarantees that the method will run without a transaction. The answer is simple. Invoking the query operation in the context of an existing transaction will cause data to be read from the database transaction log (in other words, updated data), whereas running without a transaction scope will case the query to read unchanged data from the table. For example, if you were to insert a new trade order into the `TRADE` table and subsequently (in the same transaction) retrieve a list of all trade orders, the uncommitted trade would appear in the list. However, if you were to use something like the `NotSupported` transaction attribute instead, it would cause the database query to read from the table, not the transaction log. Therefore, in the previous example, you would not see the uncommitted trade. This is not necessarily a bad thing; it depends on your use case and business logic.

The `NotSupported` transaction attribute specifies that the method being called will not use or start a transaction, regardless if one is present. If the `NotSupported` transaction attribute is specified for `methodA()` and `methodA()` is invoked in context of a transaction, that transaction is suspended until `methodA()` ends. When `methodA()` ends, the original transaction is then resumed. There are only a few use cases for this transaction attribute, and they primarily involve database stored procedures. If you try to invoke a database stored procedure within the scope of an existing transaction context and the database stored procedure contains a `BEGIN TRANS` or, in the case of Sybase, runs in unchained mode, an exception will be thrown indicating that a new transaction cannot be started if one already exists. (In other words, nested transactions are not supported.) Almost all containers use the Java Transaction Service (JTS) as the default transaction implementation for JTA. It's JTS — not the Java platform per se — that doesn't support nested transactions. If you cannot change the database stored procedure, you can use the `NotSupported` attribute to suspend the existing transaction context to avoid this fatal exception. The impact, however, is that you no longer have atomic updates to the database in the same LUW. It is a trade-off, but it can get you out of a difficult situation quickly.

The `Never` transaction attribute is perhaps the most interesting of all. It behaves the same as the `NotSupported` transaction attribute with one important difference: if a transaction context exists when a method is called using the `Never` transaction attribute, an exception is thrown indicating that a transaction is not allowed when you invoke that method. The only use case I have been able to come up with for this transaction attribute is for testing. It provides a quick and easy way of verifying that a transaction exists when you invoke a particular method. If you use the `Never` transaction attribute and receive an exception when invoking the method in question, you know a transaction was present. If the method is allowed to execute, you know a transaction was not present. This is a great way of guaranteeing that your transaction strategy is solid.

Transaction strategies

The transaction models described in this article form the basis for the transaction strategies I am about to introduce. It is important to understand fully the differences among the models and how they work before you jump into building a transaction strategy. The primary transaction strategies that can be used in most business-application scenarios are:

- Client Orchestration transaction strategy
- API Layer transaction strategy
- High Concurrency transaction strategy
- High-Speed Processing transaction strategy

I'll summarize each of these strategies here and discuss them in detail in subsequent articles in this series.

The *Client Orchestration* transaction strategy is used when multiple server-based or model-based calls from the client layer fulfill a single unit of work. The client layer in this regard can refer to calls made from a Web framework, portal application, desktop system, or in some cases, a workflow product or business process management (BPM) component. In essence, the client layer owns the processing flow and "steps" needed to complete a particular request. For example, to place a trade order, assume you need to insert the trade into the database and then update the customer's account balance to reflect the trade's value. If the application's API layer is too fine-grained, you have to invoke both methods from the client layer. In this scenario, the transactional unit of work must reside in the client layer to ensure an atomic unit of work.

The *API Layer* transaction strategy is used when you have coarse-grained methods that act as primary entry points to back-end functionality. (Call them *services* if you would like.) In this scenario, clients (be they Web-based, Web services based, message-based, or even desktop) make a single call to the back end to perform a particular request. Using the trade-order scenario from the preceding paragraph, in this case you would have a single entry-point method (called `processTrade()` for example) that the client layer calls. This single method would then contain the orchestration necessary to insert the trade order and update the account. I've given this strategy its name because in most cases, back-end processing functionality is exposed to client applications through the use of interfaces or an API. This is one of the most common transaction strategies.

The *High Concurrency* transaction strategy, a variation of the API Layer transaction strategy, is used for applications that cannot support long-running transactions from the API layer (usually because of performance or scalability needs). As the name implies, this strategy is used primarily in applications that support a high degree of concurrency from a user perspective. Transactions are fairly expensive in the Java platform. Depending on the database you are using, they can cause locks in the database, hold up resources, slow down an application from a throughput standpoint, and in some cases even cause deadlocks in the database. The main idea behind this transaction strategy is to shorten the transaction scope so that you minimize the locks in the database while still maintaining an atomic unit of work for any given client request. In some cases, you may need to refactor your application logic to support this transaction strategy.

The *High-Speed Processing* transaction strategy is perhaps the most extreme of the transaction strategies. You use it when you need to get the absolute fastest possible processing time (and hence throughput) from your application and still maintain some degree of transactional atomicity in your processing. Although this strategy introduces a small amount of risk from a data-integrity and consistency standpoint, if implemented correctly, it is the fastest possible transaction strategy in the Java platform. It is also perhaps the most difficult and cumbersome transaction strategy to implement out of the four introduced here.

Conclusion

As you can see from this overview, developing an effective transaction strategy is not always a straightforward task. Many considerations, options, models, frameworks, configurations, and techniques go into solving the data-integrity and consistency problem. In my many years working on applications and with transactions, I've found that although the combinations of models, options, settings, and configurations can seem mind-numbing and quite overwhelming, in reality only a few combinations of options and settings make sense in most use cases. The four transaction strategies I have developed and will be discussing in detail in subsequent articles in this series should cover most of the scenarios you are likely to encounter in the Java platform for business-application development. One word of caution though: These strategies are not simple "silver bullet" drop-in solutions. In some cases, source-code refactoring or application redesign may be necessary to implement some of them. When situations like that come up, you simply need to ask yourself, "How important is the integrity and consistency of my data?" In most cases, the refactoring efforts pale in comparison to the risks and costs associated with bad data.

Related topics

- *Transaction strategies* series (Mark Richards, developerWorks): Be sure to read all the installments in this series.
- *Java Transaction Design Strategies* (Mark Richards, C4Media Publishing, 2006): This book offers an in-depth discussion of transactions on the Java platform.
- *Java Transaction Processing* (Mark Little, Prentice Hall, 2004): This book is another good reference on transactions.
- Chapter 9. Transaction management: More information about Spring transaction processing can be found in this section of the Spring Framework 2.5 documentation.
- Enterprise JavaBeans 3.0 resource site: You can find documentation for the EJB 3.0 specification here.

© Copyright IBM Corporation 2009
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)