Transaction strategies: The High Performance strategy

Learn how to implement a transaction strategy for high-performance applications

Skill Level: Intermediate

Mark Richards
Director and Sr. Technical Architect
Collaborative Consulting, LLC

22 Jul 2009

In this final installment, *Transaction strategies* series author Mark Richards describes how to implement a transaction strategy in the Java™ platform for high-performance applications. Your application can maintain fast processing times while still supporting some degree of data integrity and consistency — but you need to be aware of the trade-offs involved.

So far in this series, you've learned how to implement three transaction strategies:

- The API Layer strategy, suitable for business applications with a course-grained API layer
- The Client Orchestration strategy, suitable for business applications with a more fine-grained API layer
- The High Concurrency strategy, suitable for applications that have a high concurrent-user load

In this final installment, I'll introduce a transaction strategy that's useful for applications with very high-performance requirements. Like the High Concurrency strategy, the High Performance strategy involves trade-offs to consider.

About this series

© Copyright IBM Corporation 2009. All rights reserved.

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 series, Mark Richards is your guide to designing an effective transaction strategy for use cases ranging from simple applications to high-performance transaction processing.

Transactions are necessary to ensure a high degree of data integrity and consistency. But transactions are also expensive; they consume valuable resources and can slow down an application. When you have a high-speed application for which every millisecond counts, you can maintain ACID (atomicity, consistency, isolation, and durability) properties to *some* extent by implementing the High Performance transaction strategy. As you'll see in this article, the High Performance strategy is not as robust as the other transaction strategies, and it isn't the best choice for all use cases involving high-performance applications. But there are certainly times when this strategy will allow you to maintain the fastest possible processing time and still support some degree of data integrity and consistency.

Local transactions and compensation frameworks

Local transactions in EJB 3
To use local transactions with EJB 3 session beans, you use the `@TransactionManagement(TransactionManagementType.BEAN)` annotation at the beginning of a session bean to tell the container not to manage transactions. This annotation prevents the container's transaction manager from assuming control of your transaction processing.

From a data-persistence standpoint, the fastest way to perform a database update operation is to use *local transactions* combined with database stored procedures. Local transactions (sometimes referred to as *database transactions*) are transactions that are managed by the database rather than the container environment. You don't need to code any transaction logic in your application (such as the `@Transactional` annotation in Spring or the `@TransactionAttribute` annotation in EJB 3).

Stored procedures are fast because they are precompiled and reside on the database server. They're not *required* for the High performance strategy, and their effectiveness and performance have occasioned some interesting arguments (see the "So, are Database Stored Procedures Good or Bad?" link in Resources). Using stored procedures can reduce an application's portability, add complexity, and lower overall agility. But they are generally faster than Java-based JDBC SQL statements, and when performance matters more than portability and maintenance they are a good option. That said, you can certainly use any JDBC-based framework with plain SQL if you want.

© Copyright IBM Corporation 2009. All rights reserved.

If local transactions are so fast, then why doesn't everyone use them? The main reason is that unless you use a technique such as connection passing, you cannot maintain traditional ACID transaction properties. With local transactions, database update operations are treated as individual units of work rather than a collective whole. Another limitation of local transactions is that you cannot use them with traditional object-relational mapping (ORM) frameworks such as Hibernate, TopLink, or OpenJPA. You are restricted to JDBC-based frameworks such as iBATIS or Spring JDBC (see Resources), or your own home-brewed data-access object (DAO) framework.

Connection passing

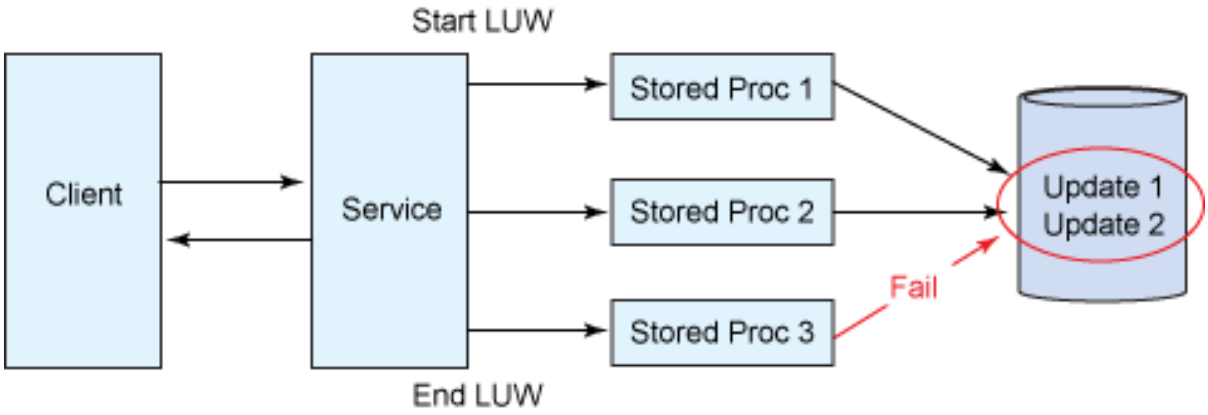
Connection passing is a technique whereby, in the absence of any sort of robust container-based transaction manager, you set the `autocommit` flag on the `Connection` object to `false` and pass the database connection around between method calls. Once you are done with the connection, you execute a `commit()` on the `Connection` object to commit the changes, or a `rollback()` to back out of the changes. Using connection passing is generally a bad idea, mainly because you are trying to do what container-based transaction managers do, but with less efficiency and greater risk of errors. If you find you are using connection passing, then you should switch to either the Programmatic or Declarative transaction model.

The High Performance strategy is based on the use of local transactions. But wait — if the Local Transaction model doesn't support basic ACID properties, then how can a transaction strategy based on this model possibly be a good strategy? The answer is that the High Performance strategy leverages the Local Transaction model in conjunction with a *compensation framework*. Each individual database update operation exists independently, but in the event of an error, the compensation framework ensures that the individual transactions are reversed, thereby maintaining both atomicity and consistency.

Figure 1 illustrates what happens when you use local transactions without a compensation framework. Notice that when the third stored procedure fails, the logical unit of work (LUW) ends, and the database is left in an inconsistent state with only two of the three updates applied:

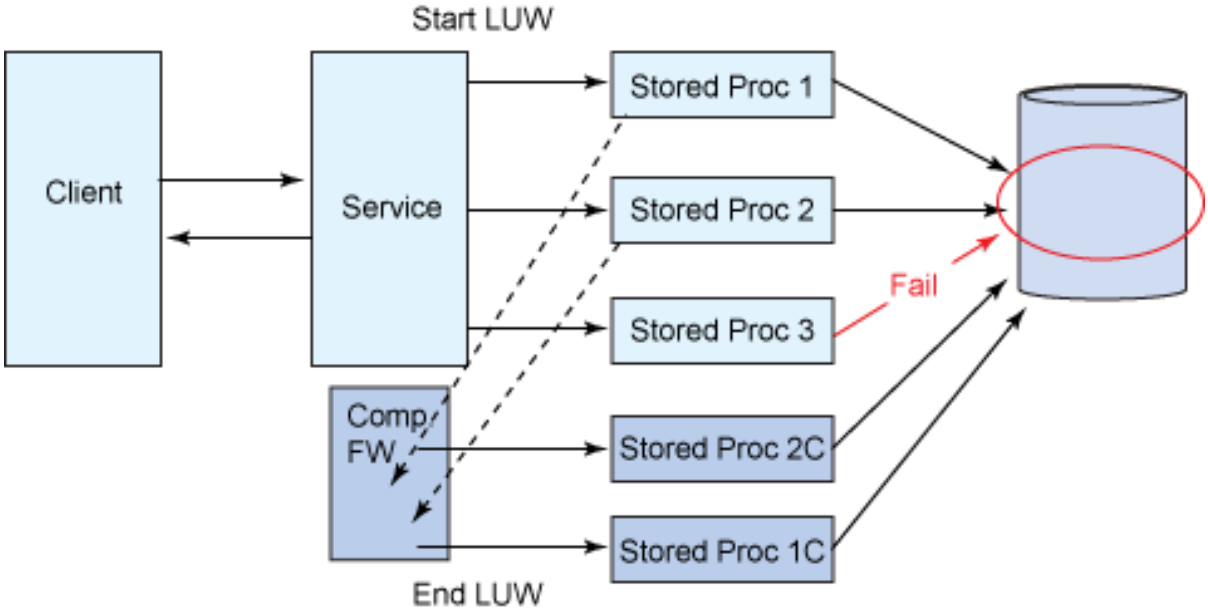
Figure 1. Local transactions without a compensation framework

© Copyright IBM Corporation 2009. All rights reserved.



With a compensation framework in place, when an error occurs the successful transactions are reversed, resulting in a consistent database. Figure 2 illustrates what happens (conceptually) when the same error occurs as in Figure 1. Notice that once the stored procedure returns successfully it registers with the compensation framework. When the third stored procedure fails, it triggers an event telling the compensation framework to reverse everything included in this *compensation scope*.

Figure 2. Local transactions with a compensation framework



This technique is commonly referred to as *relaxed ACID*. It is a typical transaction solution for such use cases as long-running transactions in a service-oriented architecture using Business Process Execution Language (BPEL) for process choreography or for using Web services. In such cases a transactional unit of work can take minutes, hours, or in some cases days to complete. It's unrealistic to assume that you can lock resources for that long. In addition, it is difficult (and sometimes impossible) to propagate a transaction across certain heterogeneous platforms or over HTTP (as in the case of Web services).

© Copyright IBM Corporation 2009. All rights reserved.

The concept of relaxed ACID can be applied to short-running transactions as well. In the case of the High Performance transaction strategy, the duration of the transaction is measured in seconds, not minutes. However, the same principles apply — in extreme high-performance situations you want to maximize database concurrency and minimize wait times and processing times. Furthermore, you want to leverage the fastest possible means for performing database update operations. This is achieved through the use of local transactions and database stored procedures. The compensation framework is there only to assist in the event of an error; it doesn't get in the way when things work correctly. Database concurrency is at its maximum, database update operations are performed the fastest way possible, and in the event of an error, everything is taken care of for you by the compensation framework. Sounds like nirvana, doesn't it? Well, unfortunately it isn't.

Trade-offs and Issues

Many trade-offs and issues are associated with this transaction strategy. Remember, it gives you the fastest possible way to perform transactions and still maintain ACID properties to *some* degree. You give up transaction isolation, robustness, and simplicity. You should use this strategy only if you cannot get the performance you need using one of the other three strategies described in this series. Compensation frameworks are complex and risky, whether you build them yourself or use one of the available open source or commercial solutions (which I'll get to a little later in this article).

Perhaps the biggest issue associated with this strategy is the general lack of robustness and data consistency that's typical of most compensation-based solutions. Because you do not have transaction isolation, each database update operation is treated as an individual unit of work. Therefore, another unit of work might take action on in-process data. If an error occurs during the LUW, it might be too late to reverse the updates; or, more typically, reversing the update can lead to cascading problems. For example, assume you are processing a very large order that will deplete the stock of that item. During processing, an event is fired (because the order is committed to the database during the LUW) to send a message automatically to the supplier to restock that item. If an error occurs during the order processing, the compensation framework will reverse the transaction, but the effect of the order (that is, the restocking message) has already been sent to the supplier, resulting in an overstock of that particular item. If traditional ACID properties were maintained, the event message to restock would not have been sent until the entire LUW of the order process was complete. This is just one of many examples illustrating how data inconsistency can occur, even when compensation frameworks are used to maintain transaction atomicity.

Certain business situations or technologies do not lend themselves to the High Concurrency transaction strategy. In particular, asynchronous processing scenarios are at high risk when compensation frameworks and relaxed ACID are used. In

© Copyright IBM Corporation 2009. All rights reserved.

some circumstances you might need to trade some level of performance for a slower ACID-based transaction strategy. Another trade-off with this strategy is that you cannot leverage ORM-based persistence frameworks, which require either the Programmatic or Declarative transaction model. This doesn't restrict you to raw JDBC coding; there are plenty of JDBC-based frameworks you can use, including iBATIS (an open source SQL mapping framework) and Spring JDBC. Or you can use your own custom DAO-based framework. The ORM restriction might require you to accept yet another trade-off — maintainability and technology selection for better performance with transaction support.

Although you maintain some level of database consistency by using compensation frameworks, a high degree of risk is associated with this strategy. Errors can occur during the transaction-reversal process, leaving your database in an inconsistent state. In this case some database update operations might be reversed while others are not, sometimes requiring manual intervention to fix the problem. For this reason, applications with few database update operations in a single LUW are good candidates for this transaction strategy. In addition, applications using this strategy typically do not have shared entity use within interleaving LUWs, meaning that it is rare for multiple users to be acting on the same entity (such as account, customer, or order) at the same time. These characteristics reduce the chance for catastrophic results stemming from inconsistency and lack of transaction isolation.

Applications suited for this particular transaction strategy are fairly robust, with errors occur infrequently (less than a 10 percent error rate). Performing transaction compensation is an expensive and time-consuming operation. If you are constantly reversing your database update operations, your system will slow down and probably perform more slowly overall than if you were to use one of the other transaction strategies. And the more you are required to perform compensatory updates, the greater the risk of database inconsistencies. Be sure to analyze your error rate before selecting this transaction strategy.

The rest of this article describes an existing compensation framework and shows a simple implementation using a custom solution to illustrate the concepts I've described.

Existing compensation frameworks

Same concept, different problem

Compensation frameworks are usually associated with long-running transactions (sometimes called L-R). They are most typically found in business process servers (such as Microsoft BizTalk Server, Oracle WebLogic Integration, Oracle BPEL Process Manager, and IBM WebSphere Process Server). Compensation frameworks are also used as solutions for solving transaction-related challenges in the Web services space. Unfortunately, these frameworks aren't suitable for implementing the High Performance transaction strategy, which is meant for short-lived decoupled activities that

© Copyright IBM Corporation 2009. All rights reserved.

need coordination, not for long-running transactions.

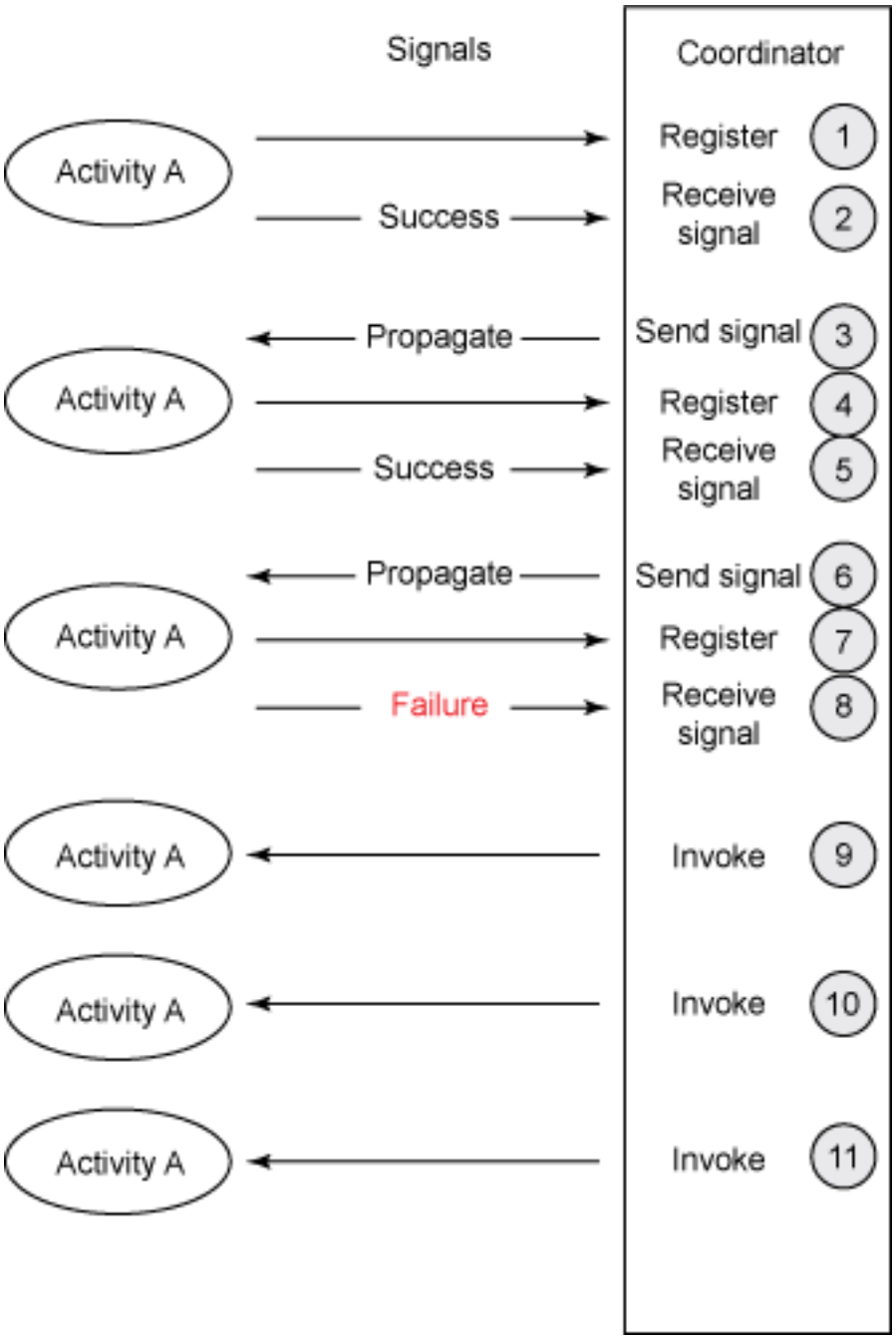
A couple of compensation frameworks are available within the Java platform: the J2EE Activity Service for Extended Transactions (JSR 95) and the JBoss Business Activity Framework (see Resources). They provide registration, messaging, and compensation trigger logic (but not the update-reversal methods themselves). Like the compensation frameworks described in the Same concept, different problem sidebar, these frameworks are generally associated with long-running transactions or Web-based requests, and as such are difficult to use with the High Performance transaction strategy. As a result, you'll most likely find yourself creating your own custom compensation framework when using this transaction strategy.

Although the J2EE Activity Service specification is primarily targeted for application server vendors, you can apply the same concepts to your own custom compensation framework. Therefore, I'll give you a brief introduction to the J2EE Activity Service in this section so you understand how compensation frameworks operate.

The J2EE Activity Service for Extended Transactions is based on the OMG Activity Service (see Resources). The J2EE Activity Service specification defines a set of interfaces and classes that coordinate and control the execution of actions within an *activity*. An activity is a set of *registered actions* (such as database update operations). Activities are controlled and coordinated by a *controller*, which is a pluggable protocol usually implemented as a third-party plug-in component. Each activity contains a *signal set* (`javax.activity.SignalSet`), which sends *signals* (`javax.activity.Signal`) to each registered action. Figure 3 shows a conceptual view of what happens when compensation is used:

Figure 3. J2EE Activity Service conceptual view

© Copyright IBM Corporation 2009. All rights reserved.



Activities must register with the controller (or more specifically a compensation manager component within the controller). Once an activity completes it sends a signal (in this case either a SUCCESS or FAILURE) to the controller. If the controller receives a SUCCESS signal from an activity, it sends a signal to the coordinator component (in this case PROPAGATE), thus invoking the next activity. Notice in step 8 of Figure 3 that a FAILURE signal is sent back to the controller. In this case the controller sends out a FAILURE signal to the coordinator, thus invoking the compensatory activities in reverse order. Although not pictured in Figure 3, the

© Copyright IBM Corporation 2009. All rights reserved.

controller also monitors the signals going back and forth between the compensatory activities and the coordinator to ensure that the reversal activities also complete successfully.

Implementing a custom compensation framework

Writing a custom compensation framework sounds like a daunting task, but it isn't overly complex — just time consuming. You can implement your own compensation framework using plain old Java code, or use more elaborate techniques. In the spirit of simplicity I'll show you a simple example using plain Java code so you understand the concept; I'll leave the creativity and fun up to you.

Whether you use an open source, commercial, or custom compensation framework, you still must provide the method, SQL, or stored procedure to invoke to reverse the database update operations. This is another reason why I like using stored procedures with the High Performance strategy. They are relatively easy to catalog, are self-contained, and they make it easier to identify (and execute) the compensatory procedure. So I'll use stored procedures in the examples I am about to present.

So that I don't bore you with unnecessary details, I am just going to assume that I already have the following stored procedures in the database ready to go:

- `sp_insert_trade` (inserts a new stock trade order into the database)
- `sp_insert_trade_comp` (reverses a trade insert by performing a delete in the database)
- `sp_update_acct` (updates the balance on an account to reflect a buy or sell of stock)
- `sp_update_acct_comp` (updates the balance on an account to the value prior to the latest update)
- `sp_get_acct` (gets an account from the database)

I'll also skip showing the DAO classes using the `CallableStatement` JDBC code so I can focus on the code that's most germane to this strategy. (See Resources for references and code for invoking stored procedures using straight JDBC). Because the implementation of the custom compensation coordinator tends to vary quite a bit and can be rather verbose, I'll show only the underlying structure and provide comments as to how to fill in the rest of the implementation code.

Depending on how you implement the strategy and what technique you use for compensatory updates, the annotations or logic you use for controlling the update-reversal operations can be in the application's API layer or its DAO layer. To

© Copyright IBM Corporation 2009. All rights reserved.

illustrate the techniques for implementing the High Performance strategy I'll use a simple stock-trading example where the logic for coordinating the compensation scope resides in the application's API layer. In this example, two activities associated with a stock trade: insert the stock trade into the database (activity 1), and update the account balance to reflect the stock trade (activity 2). Both activities are implemented in separate methods using local transactions and stored procedures. The custom compensation coordinator (`CompController`) is responsible for managing the compensation scope and reversing the activities if an error occurs.

Listing 1 illustrates the stock-trading method *without* the use of compensating transactions. The `AcctDAO` and `TradeDAO` classes referenced by the `processTrade()` method contain JDBC logic to execute the stored procedures I listed earlier. I'll skip those classes for brevity's sake.

Listing 1. Stock trade example without compensating transactions

```
public class TradingService {

   private AcctDAO acctDao = new AcctDAO();
   private TradeDAO tradeDao = new TradeDAO();

   public void processTrade(TradeData trade) throws Exception {

      try {
         //adjust the account balance
         AcctData acct = acctDao.getAcct(trade.getAcctId());
         if (trade.getSide().equals("BUY")) {
            acct.setBalance(acct.getBalance()
               - (trade.getShares() * trade.getPrice()));
          } else {
            acct.setBalance(acct.getBalance()
               + (trade.getShares() * trade.getPrice()));
          }

          //insert the trade and update the account
          long tradeId = tradeDao.insertTrade(trade);
          acctDao.updateAcct(acct);

      } catch (Exception up) {
         throw up;
      }
   }
}
```

Notice in Listing 1 the lack of transaction management (no programmatic or declarative transaction annotations or code). If an error occurs during the `updateAcct()` method, the trade inserted by the `insertTrade()` method is not rolled back, resulting in an inconsistent database. Although this code is fast, it does not support ACID transaction properties.

To apply the High Performance transaction strategy, you first need to create (or use) a compensation framework to keep track of the activities and reverse them in the event an error occurs. Listing 2 illustrates a simple example of a custom

© Copyright IBM Corporation 2009. All rights reserved.

compensation coordinator that outlines the steps needed for creating your own custom compensation framework:

Listing 2. Custom compensation framework example

```
public class CompController {

   //contains activities and the callback method to reverse the activity
   private Map compensationMap;

   //contains a list of active compensation scopes and activity sequence numbers
   private Map compensationScope;

   public CompController() {
      //load compensation map containing callback classes and methods
   }

   public void startScope(String compId) {
      //send jms start message containing compId as JMSXGroupId
   }

   public void registerAction(String compId, String action, Object data) {
      //send jms data message containing sequence number and data
      //using compId as JMSXGroupID.
      //CompController manages sequence number internally using the
      //compensationScope buffer and stores in JMSXGroupSeq message field
   }

   public void stopScope(String compId) {
      //consume and remove all messages having compId as the JMSXGroupID
      //without taking action
      //remove compId entries from compensationScope buffer
   }

   public void compensate(String compId) {
      //consume all messages having compId as the JMSXGroupID and process in
      //reverse order
      //using the compensation map and reflection to invoke reversal methods
      //remove compId entries from compensationScope buffer
   }
}
```

The `compensationMap` attribute contains a preloaded list of all activities (by name) and the corresponding class and method (by name) of the reversing activity. The contents for this example might contain the following entries: `{"insertTrade", "TradeDAO.insertTradeComp"}` and `{"updateAcct", "AcctDAO.updateAcctComp"}`. The `compensationScope` attribute contains a list of active compensation scopes by `compId` and the activities that have been registered so far. This buffer is used to get the next sequence number used by the `registerAction()` method. The rest of the methods are fairly self-explanatory.

Notice I am using Java Message Service (JMS) messaging for the compensation coordinator implementation. I chose this technique mainly because it provides a way of guaranteeing (though the use of persistent messages and guaranteed delivery) that in the event of a failure during compensation, the transactions that could not be rolled back are still in the JMS queue and can be picked up and executed by another thread. JMS messaging also allows for the possibility of an asynchronous activity registration and compensation processing, further speeding up the application

© Copyright IBM Corporation 2009. All rights reserved.

source code. Of course, keeping the compensation information in memory would significantly speed up the processing but could result in further database inconsistencies if the compensation coordinator were to fail.

The source code example in Listing 3 illustrates the technique of applying a custom compensation framework to the original application source code in Listing 1:

Listing 3. Stock trade example with compensation framework

```
public class TradingService {

   private CompController compController = new CompController();
   private AcctDAO acctDao = new AcctDAO();
   private TradeDAO tradeDao = new TradeDAO();

   public void processTrade(TradeData trade) throws Exception {

      String compId = UUID.randomUUID().toString();
      try {
         //start the compensation scope
         compController.startScope(compId);

         //get the original account values and set the acct balance
         AcctData acct = acctDao.getAcct(trade.getAcctId());
         double oldBalance = acct.getBalance();
         if (trade.getSide().equals("BUY")) {
            acct.setBalance(acct.getBalance()
               - (trade.getShares() * trade.getPrice()));
         } else {
            acct.setBalance(acct.getBalance()
               + (trade.getShares() * trade.getPrice()));
         }

         //insert the trade and update the account
         long tradeId = tradeDao.insertTrade(trade);
         compController.registerAction(compId, "insertTrade", tradeId);

         acctDao.updateAcct(acct);
         compController.registerAction(compId, "updateAcct", oldBalance);

         //close the compensation scope
         compController.stopScope(compId);

      } catch (Exception up) {
         //reverse the individual database operations
         compController.compensate(compId);
         throw up;
      }
   }
}
```

In reviewing Listing 3, notice that when defining the transactional unit of work you first start the compensation scope using the `startScope()` method. Then, you must save the original balance so that you can pass it to the coordinator when registering the activity. Once an activity completes, you then register that activity using the `registerAction()` method. This tells the compensation coordinator that a database update operation has successfully completed and needs to be added to the list of possible compensation activities. If the entire LUW ends successfully, then you invoke the `stopScope()` method, which removes all references from the

© Copyright IBM Corporation 2009. All rights reserved.

compensation coordinator. However, if an exception occurs, you invoke the `compensate()` method, which takes care of reversing whatever activities have been committed to the database.

The source code in Listings 2 and 3 is certainly far from production-ready, but it does illustrate the techniques involved with building a compensation framework of your own. Your custom compensation framework could use custom annotations, aspects (or interceptors), or even your own custom-compensation domain-specific language (DSL; see Resources) to make the code even more intuitive. Also, you don't need to use JMS asynchronous messaging for the compensation framework, but I find it useful for addressing issues surrounding compensation failures.

Conclusion

Whether or not you choose to use the High Performance strategy boils down to trade-offs. Many risks are associated with this transaction strategy, and it is complex to implement. However, if performance is your number one priority, *and* your application is already fairly robust and error-free, then this is a suitable strategy for guaranteeing at least some level of database integrity and consistency while not adversely affecting performance. Would I recommend this type of solution if performance were not your primary concern? Certainly not. You should always try to use traditional ACID properties in your application. However, if you are willing to trade some level of database consistency and data integrity for performance, then you should consider the High Performance strategy.

In this series, I've shown you some of the pitfalls and challenges associated with transaction processing and introduced four transaction strategies you can use to build a robust transaction solution for your application. Transaction processing may seem simple from the outside, but when you start to apply it to various business application scenarios it can get quite complex. My goal in this series was to reduce that complexity and bring to light techniques and strategies for simplifying what is perceived to be a challenging task — maintaining a high degree of data integrity and consistency. I hope these transaction articles have provided you with the knowledge needed to improve the robustness of your applications and data from a transaction-processing standpoint.

© Copyright IBM Corporation 2009. All rights reserved.

Resources

Learn

- *Java Transaction Design Strategies* (Mark Richards, C4Media Publishing, 2006): This book offers an in-depth discussion of transactions on the Java platform.
- *Java Message Service* (Mark Richards, O'Reilly, 2009): This book is a great reference for implementing your own custom compensation framework using JMS messaging.
- J2EE Activity Service for Extended Transactions: View the JSR 95 specification, and check out the JSR 95 (J2EE Activity Service) Reference Implementation (RI) and Technology Compatibility Kit (TCK) on developerWorks.
- "A comparison of Web services transaction protocols" (Mark Little and Thomas Freund, developerWorks, October 2003): Read a description of some of the challenges facing extended transactions.
- "So, are Stored Procedures Good or Bad?" (Roland Bouman, O'Reilly, March 2007): This article is an interesting discussion about whether stored-procedure performance really measures up to what's perceived.
- "BizTalk Server 2006: The Compensation Model" (Charles Young, December 2006): This blog entry talks about compensating transactions within Microsoft's BizTalk Server. Although not related directly to the Java platform, it nevertheless gives a good description of some of the issues and functionality of the transaction compensation model.
- "OMG Activity Service: Brief Overview" (Marek Procházka, ObjectWeb, June 2002): The J2EE Activity Service is based on the OMG Activity Service specification.
- Domain-specific language: Read about DSLs in this Wikipedia article.
- Learn more about invoking stored procedures through JDBC:
  - Calling stored procedures using `CallableStatement` methods
  - `CallableStatement`
  - Use CallableStmts to call a stored procedure
- *Java Transaction Processing* (Mark Little, Prentice Hall, 2004): This is another good reference on transactions and transaction processing on the Java platform.
- "Java theory and practice: Understanding JTS" (Brian Goetz, developerWorks, 2002): Get a handle on the basics of Java EE transaction processing in this

© Copyright IBM Corporation 2009. All rights reserved.

three-article series.

- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- iBATIS: iBATIS is a great enterprise-scale SQL mapping framework that can be used with the High Performance strategy.
- Spring JDBC: This JDBC-based framework is an option to investigate for use with the High Performance strategy.
- Business Activity Framework: Learn more about this JBoss framework.

Discuss

- Get involved in the My developerWorks community.

About the author

Mark Richards
Mark Richards is a Director and Senior Technical Architect at Collaborative Consulting, LLC, a Boston-based architecture and business consulting firm. He is the author of the 2nd edition of *Java Message Service* (O'Reilly, 2009) and of *Java Transaction Design Strategies* (C4Media Publishing, 2006). He is also a contributing author of several other books, including *97 Things Every Software Architect Should Know* (O'Reilly, 2009), *NFJS Anthology Volume 1* (Pragmatic Bookshelf, 2006) and *NFJS Anthology Volume 2* (Pragmatic Bookshelf, 2007). Mark has architect and developer certifications from IBM, Sun, The Open Group, and BEA. He is a regular speaker for the NFJS Symposium Series and speaks at other conferences and user groups throughout the world.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

© Copyright IBM Corporation 2009. All rights reserved.