Transaction strategies: The Client Orchestration strategy

Learn how to implement a robust client-based transaction strategy

Mark Richards May 19, 2009

Sometimes an application's presentation layer must handle the coordination of multiple API layer method calls to complete a single transactional unit of work. In this article, *Transaction strategies* series author Mark Richards describes the Client Orchestration transaction strategy and explains how to implement it in the Java™ platform.

View more content in this series

If you are following along in this series, you know by now that you need an effective and robust transaction strategy to ensure a high level of data consistency and data integrity, regardless of the language, environment, framework, or platform you are using. In this article, I'll describe the Client Orchestration transaction strategy, which I introduced briefly in "Models and strategies overview." As the name suggests, this strategy is used when the application's client layer must make one or more calls to the API layer to complete a single transactional unit of work. I'll use the EJB 3.0 specification in my code examples; the concepts are the same for the Spring Framework and Java Open Transaction Manager (JOTM).

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

Sometimes applications are written with a fine-grained API layer, requiring the client to make multiple calls to the API layer for a single logical unit of work (LUW). This might be because of complex and diverse client requests that cannot be aggregated with a coarser-grained API model, or it might simply result from poor application design. Regardless of the reason, when the percentage of multiple API-layer method calls from the client exceeds a reasonable amount to refactor to a single API-layer call, it's time to abandon the simpler API Layer strategy and adopt the Client Orchestration transaction strategy.

Basic structure

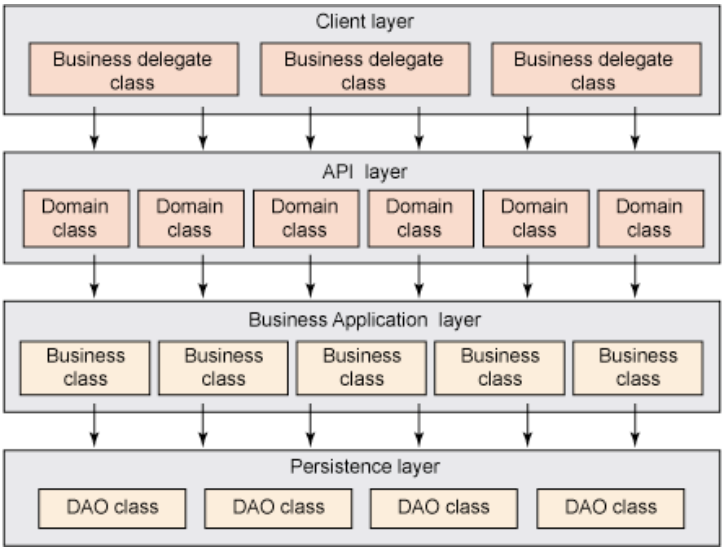
In "The API Layer strategy," I outlined two golden rules for building a transaction strategy:

- The method that starts the transaction is designated as the *transaction owner*.
- Only the transaction owner can roll back the transaction.

I mention these rules again because they also apply to the Client Orchestration transaction strategy. Regardless of where the method that starts the transaction resides, it's important that the transaction owner be the only method to manage the transaction and perform the commit or rollback.

Figure 1 illustrates a typical logical application layer stack for most Java applications:

Figure 1. Architecture layers and transaction logic



The architecture in Figure 1 implements the Client Orchestration transaction strategy. The classes containing transaction logic are shaded in red. Notice in this strategy that both the client layer and the API layer contain transaction logic. The client layer controls the transaction scope. It's here that transactions are started, committed, and rolled back. The API-layer methods contain transaction directives instructing the transaction manager to incorporate and use the transaction scope started by the client layer. The business layer and persistence layer contain no transaction logic, meaning that these layers do not start, commit, or roll back a transaction, nor do they contain transaction annotations such as the `@TransactionAttribute` annotation found in EJB 3.0.

Don't get hung up on the fact that Figure 1 shows four layers. Your application architecture could have more layers or fewer. You might combine the presentation and domain layers in a single WAR file, or your domain classes might be in a separate EAR file. You might have the business logic located in the domain classes as one layer rather than two. None of that matters with respect to how the transaction strategy works or how it is implemented.

This transaction strategy is well-suited for applications that have a complex and fine-grained API layer. These applications — generally known as *chatty* — require several calls to the API layer to

fulfill a single LUW. The Client Orchestration transaction strategy does not carry the same single API-layer call restriction as the API Layer transaction strategy: you could make one call to the API layer from the client layer or many calls for each LUW. However, from an application-architecture standpoint, this transaction strategy is more restrictive than the other transaction strategies in that the client layer must be able to start a transaction and propagate it to the API layer. This means you cannot use a Java Message Service (JMS) messaging client, Web services client, or a non-Java client. In addition, the communication protocol between the client layer and the API layer (if any) must support the propagation of a transaction (for example, RMI over Internet Inter-Orb Protocol [RMI-IIOP]; see Related topics).

I am not promoting the use of a fine-grained API-layer chatty application architecture; rather, I am saying that *if* you have an application that is chatty and cannot be refactored, then the Client Orchestration transaction strategy is probably the right one to apply.

Strategy rules and characteristics

The following rules and characteristics apply to the Client Orchestration transaction strategy:

- Only methods in the client layer and the API layer of the application architecture should contain transaction logic. No other methods, classes, or components should contain transaction logic (including transaction annotations, programmatic transaction logic, and rollback logic).
- The client-layer methods are the only methods that are responsible for starting, committing, and rolling back the transaction.
- The client-layer method that starts the transaction is considered the *transaction owner*.
- In most cases, programmatic transactions are required in the client layer, meaning that you must programmatically obtain a transaction manager and code the begin, commit, and rollback logic. The exception to this rule is if the client business delegate in the client layer controlling the transaction scope is managed as a Spring bean by the Spring Framework. In this case, you can use the declarative transaction model provided by Spring.
- Because you cannot programmatically pass a transaction context, the API layer must use the declarative transaction model, meaning that the container manages the transaction. You only need to specify the transaction attributes (no rollback code or rollback directives!).
- All public write methods (insert, update, and delete) in the API layer should be marked with a transaction attribute of `MANDATORY`, indicating that a transaction is needed but the transaction context must be established before the method is invoked. Unlike the `REQUIRED` attribute, the `MANDATORY` attribute will *not* start a transaction if one does not exist, but rather will throw an exception indicating that a transaction is required.
- No public write methods (insert, update, and delete) in the API layer should contain rollback logic, regardless of the type of exception thrown.
- All public read methods in the API layer by default should be marked with a transaction attribute of `SUPPORTS`. This ensures that the read method is included in a transaction scope if it is invoked within that scope's context. Otherwise, it will run without a transaction context, with the assumption that it is the only method being invoked within the logical unit of work (LUW). I am making the assumption here that the read operation (as an entry point to the API layer) is not in turn invoking write operations on the database.

- The transaction context from the client layer will propagate to the API layer methods and all methods invoked under the API layer.
- If the client layer is making remote calls to the API layer, the client layer must use a protocol and transaction manager that supports the propagation of a transaction context (such as RMI-IIOP).

Limitations and restrictions

As I said earlier, one of the biggest restrictions with this transaction strategy is that the client layer must be able to start a transaction and propagate it to the API layer. This means that the protocol used to communicate between the client layer and the API layer, as well as the type of client, play an important role in the application architecture. For example, you cannot use a Web service client or a JMS client with this strategy, nor can you rely on HTTP communications between the client layer and the API layer; the protocol used between the two layers must be able to support the propagation of a transaction.

Another limitation of this strategy, unlike the API Layer transaction strategy, is that you cannot "cheat" and introduce it into your application architecture incrementally. With the API Layer transaction strategy, it's not the end of the world if you start a transaction at the client layer during a refactoring effort. The impact of doing this with the API Layer transaction strategy is that the client layer will not be able to take corrective action on an exception, and you will not be able to roll back a transaction if it has already been rolled back in the API layer. Sloppy, but not devastating.

However, with the Client Orchestration transaction strategy, because the API layer uses a transaction attribute of `MANDATORY` and does not contain transaction-rollback logic, the client layer methods must start a transaction. Changing the API-layer methods to `REQUIRED` and adding rollback logic introduces the same problems outlined in the Limitations and Restrictions section in "The API layer strategy." Furthermore, using `REQUIRED` in the API layer methods means that a transaction *could* be started by the API layer, thereby violating the main principles of the Client Orchestration transaction strategy.

Transaction strategy implementation

Implementation of the Client Orchestration transaction strategy is fairly straightforward, but because it involves both the client layer and the API layer of the architecture, I'll show the transaction logic and strategy implementation for both layers' methods.

Recall from the Strategy rules and characteristics section that in most cases, the client layer needs to use *programmatic transactions* unless specifically running under a Spring context as a Spring-managed bean. Because I am using EJB3 for my implementation examples, I'll show you the implementation using programmatic transactions. You can refer to "Models and strategies overview" or the Spring Framework documentation for an example of how to use programmatic transactions in Spring (see Related topics).

Also, if you are running an external client, make sure that your transaction manager supports the propagation of a transaction across JVMs. For my examples, I am using JBoss 4.2.0 with EJB 3.0

using the Java Persistence API (JPA) with Hibernate 3.2.3, running with MySQL 5.0.51b using the InnoDB engine. This environment (specifically JBoss) supports the propagation of a client transaction across multiple JVMs (using RMI-IIOP).

I'll start with the read operations because those are the easiest. For database read operations originating at the client layer, you don't need to do a thing in the client code from a transaction standpoint, because a transaction is not required for a database read operation (see the Never say never sidebar in "Understanding transaction pitfalls"). However, as with the API Layer transaction strategy, you will want to set the API layer read-operation methods to `SUPPORTS` to ensure that the read method is included in a transaction scope if it is invoked within that scope's context.

Listing 1 illustrates a simple client-layer method invoking a read operation. Notice that no transaction logic is needed in the `getTrade()` read method:

Listing 1. Read operation — client layer

```
package com.trading.client;

import javax.naming.InitialContext;
import com.trading.common.TradeData;
import com.trading.common.TradingService;

public class TradingClient {

   public static void main(String[] args) {
      new TradingClient().getTrade();
   }

   public void getTrade() {
      try {
         InitialContext ctx = new InitialContext();
         TradingService service = (TradingService)
            ctx.lookup("TradingServiceImpl/remote");

         TradeData trade = service.getTrade(11);
         System.out.println(trade);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

The corresponding `getTrade()` API-layer read method in the `TradingServiceImpl` EJB3 stateless session bean is shown in Listing 2:

Listing 2. Read operation — API layer

```
package com.trading.server;

import javax.ejb.Stateless;
import javax.ejb.Remote;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import com.trading.common.TradeData;
import com.trading.common.TradingService;
```

```
@Stateless
@Remote(TradingService.class)
public class TradingServiceImpl implements TradingService {
   @PersistenceContext EntityManager em;

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

Notice that the `SUPPORTS` transaction attribute is used in Listing 2, indicating that if this method is called independently, it will not start a transaction, but it will use an existing transaction context if it is invoked within the context of an existing transaction.

For database update operations, the client layer, being the *transaction owner*, is responsible for obtaining a transaction manager, starting the transaction, then either committing the transaction or rolling it back based on the outcome of the operations. Typically you need to use *programmatic transactions* in the client layer. In EJB3, this is done by first establishing an `InitialContext` to the application server, then performing a lookup on the Java Naming and Directory Interface (JNDI) name of the `UserTransaction`. For JBoss, the JNDI name is `UserTransaction`. You can refer to my transaction book for a list of the JNDI names for most of the common application servers (see Related topics), or consult the documentation for the application server you are using. Once you have a `UserTransaction`, you can code the `begin()` method to start the transaction, the `commit()` method to commit the transaction, and — if an exception occurs — the `rollback()` method to roll back the transaction. Listing 3 shows the complete source code for a client-layer method making an orchestrated update request to the API layer to insert a stock trade and update the customer account:

Listing 3. Update operation — client layer

```
package com.trading.client;

import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
import com.trading.common.AcctData;
import com.trading.common.TradeData;
import com.trading.common.TradingService;

public class TradingClient {

   UserTransaction txn = null;

   public static void main(String[] args) {
      new TradingClient().placeTrade();
   }

   public void placeTrade() {
      try {
         InitialContext ctx = new InitialContext();
         TradingService service = (TradingService)
            ctx.lookup("TradingServiceImpl/remote");

         TradeData trade = new TradeData();
         trade.setAcctId(1234);
         trade.setAction("BUY");
         trade.setSymbol("AAPL");
```

```
         trade.setShares(100);
         trade.setPrice(103.45);

         txn = (UserTransaction)ctx.lookup("UserTransaction");
         txn.begin();
         service.insertTrade(trade);
         service.updateAcct(trade);
         txn.commit();
      } catch (Exception e) {
         try {
            txn.rollback();
         } catch (Exception e2) {
            e2.printStackTrace();
         }
         System.out.println("ERROR: Trade Not Placed");
         e.printStackTrace();
      }
   }
}
```

Because an update method in the client layer is always the transaction owner in the Client Orchestration transaction strategy, the public update methods in the API layer should never start a transaction. For this reason, they must use a transaction attribute of `MANDATORY`, indicating that a transaction is needed by the method but it should have been started elsewhere (such as in the client layer). Furthermore, in compliance with the second golden rule, the update methods in the API layer should not contain any transaction-rollback logic. Listing 4 shows a complete example of an EJB3 stateless session bean implementing the Client Orchestration transaction strategy for the corresponding client layer code shown in Listing 3:

Listing 4. Update operation — API layer

```
package com.trading.server;

import javax.ejb.Remote;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import com.trading.common.AcctData;
import com.trading.common.TradeData;
import com.trading.common.TradingService;

@Stateless
@Remote(TradingService.class)
public class TradingServiceImpl implements TradingService {
   @PersistenceContext EntityManager em;

   @TransactionAttribute(TransactionAttributeType.MANDATORY)
   public TradeData insertTrade(TradeData trade) throws Exception {
      trade.setStage("PLACED");
      em.persist(trade);
      return trade;
   }

   @TransactionAttribute(TransactionAttributeType.MANDATORY)
   public void updateAcct(TradeData trade) throws Exception {
      AcctData acct = em.find(AcctData.class, trade.getAcctId());
      if (trade.getAction().equals("BUY")) {
         acct.setBalance(acct.getBalance() - (trade.getShares() * trade.getPrice()));
```

```
      } else {
         acct.setBalance(acct.getBalance() + (trade.getShares() * trade.getPrice()));
      }
   }
}
```

As you can see from Listing 4, if an exception is thrown in either of the update methods, it is the client layer method's responsibility to do the necessary transaction rollback. As you can also see from Listing 4, with this strategy the client layer *must* be able to start and propagate a transaction; otherwise, you will get a `javax.ejb.EJBTransactionRequiredException` indicating that a transaction is needed to invoke the update methods.

Conclusion

The Client Orchestration transaction strategy is useful when most of the requests from the client layer require multiple calls to the API layer to complete a single LUW. Be careful though — implementing this strategy places restrictions on the application architecture in terms of what kind of clients the architecture will support and the communication protocol used between the client layer and the API layer. Another disadvantage of this transaction strategy is that using programmatic transactions in the client layer always leaves room for "programmer error," not to mention the fact that client-side developers now must learn about the Java Transaction API (JTA) and corresponding transaction logic.

Don't be tempted to mix the Client Orchestration strategy and API Layer strategy in the same application to try to solve every permutation in your application architecture. It simply won't work, and you'll end up with inconsistent data in your database and an overly complex design. If you do have clients that do not support transactions but you find the Client Orchestration transaction strategy seems like a good fit, you may have to do some refactoring. One of the ways out of this "hybrid" mess is to provide an "alternative API" facade layer using the API Layer transaction strategy that calls into the API layer using the Client Orchestration strategy. However, bear in mind that the calls to this alternative API must be single calls (as specified in the API Layer transaction strategy). Essentially, you are treating the alternative API as the client to the API layer. From here, you can make multiple API layer calls, because the transaction would originate at the new alternative API.

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 by the article author offers an in-depth discussion of transactions on the Java platform.
- Java EE APIs and Docs: You can find documentation for the EJB 3.0 specification here.
- *Enterprise JavaBeans 3.0 (5th Edition)* (Bill Burke and Richard Monson-Haefel, O'Reilly Media, 2006): This is an excellent book for coming up to speed quickly with EJB 3.0.
- *Java Transaction Processing* (Mark Little, Prentice Hall, 2004): This book is another good reference on transactions.
- Chapter 9. Transaction management: You can find more information about Spring transaction processing in this section of the Spring Framework 2.5 documentation.
- Java Open Transaction Manager: JOTM is an open source stand-alone transaction manager implementing the XA protocol and compliant with the JTA APIs. It's useful if you're using a stand-alone Tomcat or Jetty container or a stand-alone application requiring transaction support.
- Java RMI over IIOP: Learn more about RMI-IIOP.