Transaction strategies: The API Layer strategy

Learn how to implement a simple yet robust transaction strategy

Mark Richards April 14, 2009

An effective and robust transaction strategy is critical for maintaining data consistency and integrity. The API Layer transaction strategy is easy to implement and is well-suited for most business applications. Using examples from the Enterprise JavaBeans (EJB) 3.0 specification, *Transaction strategies* series author Mark Richards explains what this transaction strategy is and how to implement it in the Java™ platform.

View more content in this series

Whether you are using a container environment with EJB 2.1 or 3.0, the Spring Framework environment, or a Web container environment such as Tomcat or Jetty with the Java Open Transaction Manager (JOTM), you still need a transaction strategy to ensure database consistency and integrity. The Java Transaction API (JTA) specifies the syntax and interfaces associated with transaction processing (see Related topics), but it doesn't describe how to put these building blocks together. Just as a construction crew needs a blueprint to build a house out of a pile of lumber, you need a strategy that describes *how* the transactional building blocks are put together.

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.

The strategy I describe in this article is the *API Layer transaction strategy*. It is the most robust, simplest, and easiest-to-implement transaction strategy. With that simplicity comes limitations and considerations that I also describe. I use the EJB 3.0 specification in my code examples; the concepts are the same for the Spring Framework and JOTM.

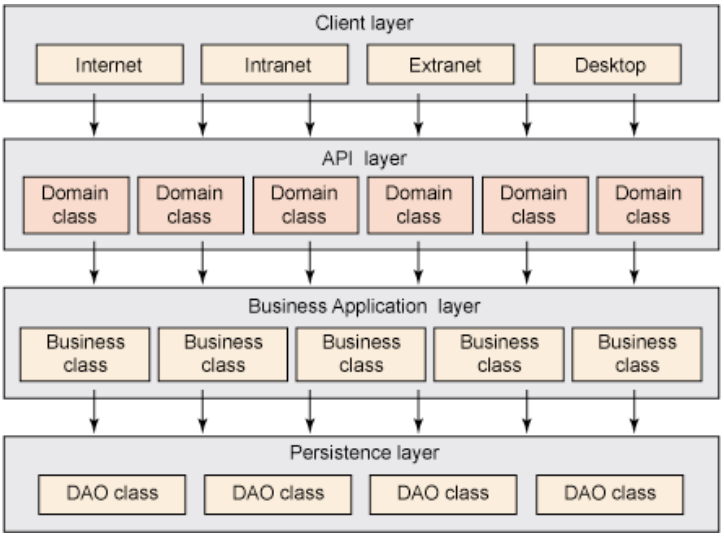
Basic structure

The API Layer transaction strategy gets its name from the fact that all transaction logic is contained in the API layer of the logical application architecture. This layer — sometimes referred to as the application's *domain layer* or *facade layer* — is the logical layer that exposes functionality to the client (or presentation layer) in the form of public methods or interfaces. I say *logical* here

because you can access the domain layer locally (by direct instantiation and invocation) or remotely via HTTP, remote method invocation (RMI), RMI over Internet Inter-Orb Protocol (RMI-IIOP) via EJB, or even through the Java Message Service (JMS).

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 API Layer transaction strategy. The classes containing transaction logic are shaded in red. Notice that these consist only of the application architecture's domain classes (the API layer). The client layer, 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. The only methods in the entire application architecture that start, commit, and roll back a transaction are public methods contained in API layer's domain classes. This is why API Layer is the most robust and easiest transaction strategy to implement.

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 coarse-grained API layer. And because the presentation layer does not contain any transaction logic (even for update requests), this strategy is well-suited for applications that must support multiple client channels, including Web service clients, desktop clients, and remote clients. But this flexibility comes with a price — namely, that the client layer is restricted to a single request for a given transactional unit of work. I'll explain why this restriction is necessary later in this article.

Strategy settings and characteristics

The following rules and characteristics apply to the API Layer transaction strategy:

- Only the public methods included in the application architecture's API layer contain transaction logic. No other methods, classes, or components should contain transaction logic (including transaction annotations, programmatic transaction logic, and rollback logic).
- All public write methods (insert, update, and delete) in the API layer should be marked with a transaction attribute of `REQUIRED`.
- All public write methods (insert, update, and delete) in the API layer should contain rollback logic to mark the transaction for rollback on a checked exception.
- All public read methods in the API layer by default should be marked with a transaction attribute of `SUPPORTS`. (See the Never say never sidebar in "Transaction strategies: Understanding transaction pitfalls.") This will ensure 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 from the API layer will propagate to all methods invoked under the *transaction owner* (as defined in the next section).
- The Declarative Transaction model is normally used for this pattern, with the assumption that the API-layer classes are managed by a Java EE container environment or by another framework such as Spring. If they are not, then you will most likely need to use the Programmatic Transaction model. (See "Transaction strategies: Models and strategies overview" for more information on these transaction models.)

If you look closely, you might notice a slight problem with this strategy, given the rules I've just listed. Because of interservice communication, one public API-layer update method can invoke another public API-layer update method. Because both update methods are public and are therefore exposed as API-layer entry points from the client-layer perspective, they contain rollback logic. However, if one public update method invokes another, the transaction owner may not have control of the rollback logic in some circumstances. For this reason care must be taken if the transaction owner resubmits the transaction or takes corrective action. In these cases you may need to refactor your orchestration or processing logic to avoid this scenario.

Limitations and restrictions

One of this transaction strategy's restrictions is that the client-layer (or presentation-layer) classes can make only a single call to the API layer for any given transactional unit of work. That makes this transaction strategy ill suited for applications that are "chatty." Unfortunately, this is an all-or-nothing proposition, and in some cases may require application refactoring (described later in this section). Let me explain why this is so important (and necessary) for this transaction strategy.

Two golden rules (the secret sauce) apply to all of the transaction strategies I'll describe throughout the *Transaction strategies* series:

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

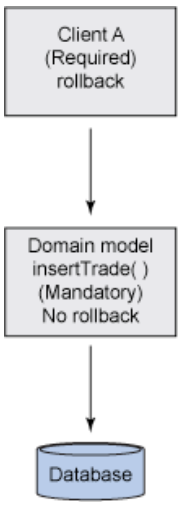
If you do not comply with these rules, the transaction strategy will not work. You will most likely end up with a problem that results in inconsistent data and poor data integrity. The second rule

is important for several reasons. First, if a method did not start a transaction, it has no business managing that transaction (for example, marking it for rollback). Second, if a lower-level method call in the chain rolls back the transaction, the transaction owner cannot take corrective action to repair and resubmit the transaction; once it has been marked for rollback, that is the only possible outcome. You cannot "unrollback" a transaction.

Back to the original point: with the API Layer transaction strategy, the client *must* not make multiple calls to the API layer in a single unit of work requiring a transaction. If the client makes multiple calls to the API for a single LUW, then the transactional unit of work must start and end at the client. In this case, the API layer methods must have a transactional attribute of `MANDATORY` without any rollback logic. Remember the golden rules: the client method invoking the API layer method is the transaction owner, and only the transaction owner should be responsible for rollback.

Consider the example illustrated in Figure 2, where a client (Client A) makes a request to the API layer (domain model) to insert a stock trade into the database:

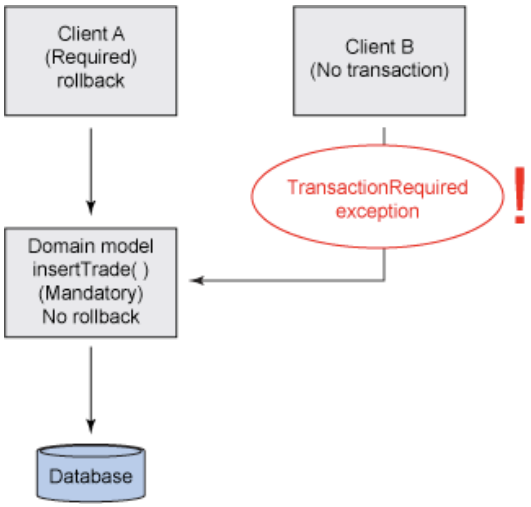
Figure 2. Use of `MANDATORY` attribute



In this case, the client is starting the transaction; hence the `REQUIRED` transaction attribute. Notice that the client also has rollback responsibility, complying with my two golden rules. The domain class properly has a transaction attribute of `MANDATORY` because the client is starting the transaction, and the domain model (`insertTrade()`) is not responsible for rollback.

This strategy is correct for the scenario illustrated in Figure 2. However, suppose you have another client application (Client B) needing to use the same domain model (`insertTrade()`), as shown in Figure 3:

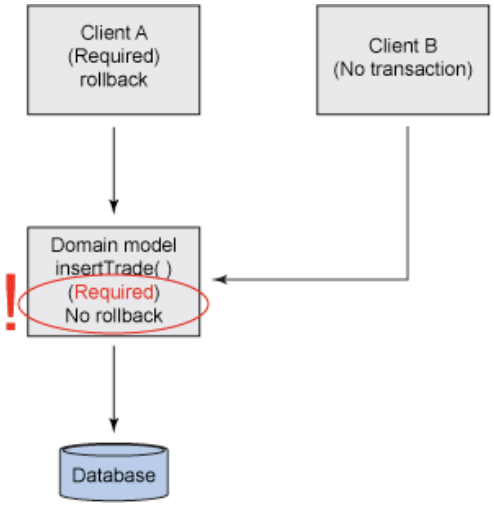
Figure 3. Nontransactional client problem



Notice here that Client B is not starting a transaction. (It could be a remote HTTP client, messaging client, or other Java application unable to use transactions.) Because the domain model method is marked `MANDATORY`, Client B will experience a `TransactionRequiredException` indicating that a transaction is required to invoke the method.

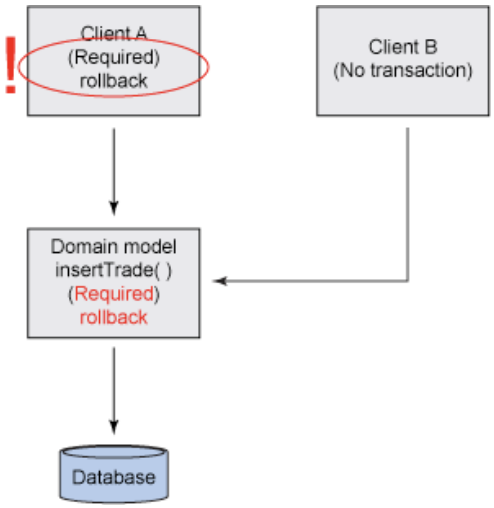
Without a proper transaction strategy in place, the way this problem is typically addressed is to change the transaction attribute on the domain model (`insertTrade()`) to `REQUIRED`. Now, if you go back to the invocation by Client A, you'll notice that you haven't broken anything; Client A starts a transaction, then passes the transaction context to the domain-model method. Because the domain-model method is now marked `REQUIRED`, it uses the existing transaction context. Notice that the domain-model method does not contain rollback logic. After the domain model is done (whether an exception occurs or not), control is passed back to the client. This all works correctly for Client A. However, if you look at Client B, you have a problem: because no transaction context is present, the domain-model method (`insertTrade()`) starts a new transaction, but in the event of a checked exception, no rollback is performed because the domain-model method is not responsible for rollback. Figure 4 illustrates this error condition:

Figure 4. Use of `REQUIRED` attribute without rollback



The way this problem is typically addressed without a proper transaction strategy in place is to add rollback logic to the domain-model method to satisfy the invocation from Client B. However, as Figure 5 illustrates, this now causes problems in Client A:

Figure 5. Client rollback problem



Not only will Client A receive an exception if it tries to roll back the transaction, but Client A cannot take corrective action because the transaction has already been marked for rollback.

Ping, pong. And the beat goes on....

This is why transaction strategies are so important, and also why they must be absolute. As I mentioned previously, you could find yourself with an application with 85 percent single-API-layer calls and 15 percent multiple-API-layer calls for LUW requests. If this is the case (or something close to it), you have two choices: don't coordinate the multiple calls within a transactional unit of work (which is a bad idea), or — preferably — refactor the multiple API calls into a single API call using an aggregate API-layer method.

To illustrate this refactoring technique, assume you have two API layer methods, `insertTrade()` and `updateAcct()`, as shown in Listing 1:

Listing 1. Multiple API layer methods

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

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public long insertTrade(TradeData trade) throws Exception {
      try {
         em.persist(trade);
         return trade.getTradeId();
      } catch (Exception up) {
         ctx.setRollbackOnly();
         throw up;
      }
   }

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void updateAcct(TradeData trade) throws Exception {
      try {
         //update account balance based on buy or sell...
      } catch (Exception up) {
         ctx.setRollbackOnly();
         throw up;
      }
   }
}
```

Assume that both of these methods can be invoked as independent operations. However, there are times, as illustrated in Listing 2, that a client can invoke both of these methods in the same LUW:

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

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

Rather than messing everything up by placing transactions at the client layer, it is better to remove the multiple-API-layer invocation by creating a new aggregate method in the `TradingServiceImpl` class and corresponding `TradingService` interface. Listing 3 shows the new aggregate method (the interface code is not shown, for brevity):

Listing 3. Adding a public aggregate method

```
@Stateless
@Remote(TradingService.class)
```

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

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public TradeData placeTrade(TradeData trade) throws Exception {
      try {
         insertTrade(trade);
         updateAcct(trade);
         return trade;
      } catch (Exception up) {
         ctx.setRollbackOnly();
         throw up;
      }
   }

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public long insertTrade(TradeData trade) throws Exception {
      ...
   }

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void updateAcct(TradeData trade) throws Exception {
      ...
   }
}
```

The refactored client code is shown in Listing 4:

Listing 4. Refactored client method making only one API-layer call

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

Notice that the client method is now making a single call to a new aggregate method in the API layer called `placeTrade()`. The `insertTrade()` and `updateAcct()` methods are still public because they can also be invoked independently of each other, regardless of the new aggregate method.

Although I'm using a simple example for illustrative purposes I don't mean to trivialize the possible complexity of this refactoring technique. In some cases — particularly with client code using Web-based objects such as `HTTPServletRequest` or `HTTPSession` — the refactoring can be complex and involve significant code changes. You need to consider the trade-off between the effort to refactor the client and server code versus the need for and importance of data integrity and consistency. That said, let me put on my pragmatic hat for a moment. For purposes of an *incremental move* toward a solid transaction strategy (such as the API Layer transaction strategy described in this article), you can add transaction logic to the client code temporarily, thereby keeping the two API-layer calls coordinated within the same transactional unit of work (making sure the API layer still has the `REQUIRED` attribute setting). However, you should understand the implications of doing this:

- You would need to use programmatic transactions in the client method (see "Transaction strategies: Models and strategies overview").

- You would need to wrap the transaction rollback in a `try/catch` block in the event the API-layer methods marked the transaction for rollback.
- You would not be able to take corrective action on an exception.
- You would be restricted to the communication protocol used between the client and API layer (for example, no HTTP, no JMS, and so on).

Just be aware that by implementing this transaction strategy incrementally, you will not have a solid and robust transaction strategy until you are done with the refactoring effort.

Transaction strategy implementation

The implementation of the API Layer transaction strategy is fairly straightforward. Because the only layer containing transaction logic is the API layer, I will only show the transaction logic in that layer's domain-model classes.

Recall from Strategy settings and characteristics that for write operations (updates, inserts, and deletes), the public API-layer methods should have a transaction attribute of `REQUIRED` and contain transaction rollback logic. And public read operations by default should have a transaction attribute of `SUPPORTS` with no rollback logic. Listing 5 below illustrates this transaction-strategy implementation:

Listing 5. Implementing the API Layer strategy

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

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public long insertTrade(TradeData trade) throws Exception {
      try {
         em.persist(trade);
         return trade.getTradeId();
      } catch (Exception up) {
         ctx.setRollbackOnly();
         throw up;
      }
   }

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

An optimized approach to implementing this strategy is to leverage the `TYPE` scope of the `@TransactionAttribute` annotation in EJB 3.0 and make all methods in the entire class by default `REQUIRED`, and override only the read operations to `SUPPORTS`. This technique is illustrated in Listing 6:

Listing 6. Optimizing the implementation of the API Layer strategy

```
@Stateless
```

```
@Remote(TradingService.class)
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class TradingServiceImpl implements TradingService {
   @PersistenceContext(unitName="trading") EntityManager em;
   @Resource SessionContext ctx;

   public long insertTrade(TradeData trade) throws Exception {
      try {
         em.persist(trade);
         return trade.getTradeId();
      } catch (Exception up) {
         ctx.setRollbackOnly();
         throw up;
      }
   }

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

The reason I suggest this approach rather than making everything `SUPPORTS` by default is that if you forget to code the `@TransactionAttribute` annotation, it's better to err on the side of having a transaction rather than not having one.

Conclusion

The API Layer transaction strategy will work fine for most business applications. It is straightforward, simple, easy to implement, and robust. You may be faced with some application refactoring to implement this strategy, but in the long run you'll find that the savings from having a high level of data consistency and integrity will more than compensate for the effort. Remember, what I described in this article is a transaction *strategy*. It involves much more than just a simple implementation task. Every developer on the team should know and understand the transaction strategy being used, be able to describe it, and — most important — be able to comply with it.

Although the API Layer transaction strategy is the most common strategy, it may not be the right one for your application. For example, there may be times when the percentage split between single API calls versus multiple API calls within a single LUW is reversed (say, 20 percent single-API-layer calls vs. 80 percent multiple-API-layer calls). In circumstances like these, you might not want to take on such a behemoth refactoring effort. That is when you would use the Client Orchestration transaction strategy, which I'll discuss in the next installment of this series. Or the duration of the transaction as implemented in the API Layer transaction strategy might be too long, resulting in database concurrency issues, reduced throughput, connection wait times, and in extreme cases, database deadlocks. These symptoms are usually found in a high-concurrency environment where an application must process a large user volume or load. In these circumstances, you would use the High Concurrency transaction strategy — the topic of this series' fifth article. Finally, you may find yourself in a standard application architecture but one that is designed for high-speed requirements where every millisecond of processing time matters. For these circumstances, you'll want to use the High Speed transaction strategy, which I'll describe in the sixth installment.

Related topics

- *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.
- "*Java theory and practice*: Understanding JTS" (Brian Goetz, developerWorks, 2002): Get a handle on the basics of Java EE transaction processing in this three-article series.