Advanced DAO programming

Learn techniques for building better DAOs

Level: Advanced

Sean C. Sullivan (dao-article@seansullivan.com)
Software Engineer
October 7, 2003

J2EE developers use the Data Access Object (DAO) design pattern to separate low-level data access logic from high-level business logic. Implementing the DAO pattern involves more than just writing data access code. In this article, Java developer Sean C. Sullivan discusses three often overlooked aspects of DAO programming: transaction demarcation, exception handling, and logging.

Contents:

DAO fundamentals
Transaction demarcation
Transaction demarcation with JDBC
Overview of JTA
Transaction demarcation with JTA
JTA methods for transaction control
Using JTA and JDBC
Choosing the best approach
Logging and DAOs
Exception handling in DAOs
Implementation example: MovieDAO
Conclusion
Resources
About the author
Rate this article

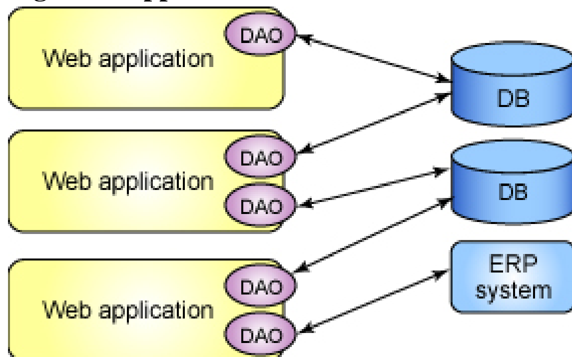
Related content:

A stepped approach to J2EE testing with SDAO
Create persistent application data with Java Data Objects
Understanding JTS -- An introduction to transactions
Subscribe to the developerWorks newsletter
developerWorks Toolbox subscription

During the past 18 months I worked with a team of talented software engineers to build custom Web-based supply chain management applications. Our applications accessed a broad range of persistent data, including shipment status, supply chain metrics, warehouse inventory, carrier invoices, project management data, and user profiles. We used the JDBC API to connect to our company's various database platforms and applied the DAO design pattern throughout the applications.

Figure 1 shows the relation between the applications and data sources:

Figure 1. Applications and data sources



Applying the Data Access Object (DAO) pattern throughout the applications enabled us to separate low-level data access logic from business logic. We built DAO classes that provide CRUD (create, read, update, delete) operations for each data source.

In this article, I'll introduce you to DAO implementation strategies and techniques for building better DAO classes. Specifically, I'll cover logging, exception handling, and transaction demarcation. You will learn how to incorporate all three in your DAO classes. This article assumes that you are familiar with the JDBC API, SQL, and relational database programming.

We'll start with a review of the DAO design pattern and data access objects.

DAO fundamentals

The DAO pattern is one of the standard J2EE design patterns. Developers use this pattern to separate low-level data access operations from high-level business logic. A typical DAO implementation has the following components:

- A DAO factory class
- A DAO interface
- A concrete class that implements the DAO interface
- Data transfer objects (sometimes called value objects)

Also in the Java zone:

Tutorials

Tools and products

Code and components

Articles

The concrete DAO class contains logic for accessing data from a specific data source. In the sections that follow you'll learn techniques for designing and implementing data access objects. See Resources to learn more about the DAO design pattern.

Transaction demarcation
The important thing to remember about DAOs is that they are transactional objects. Each operation performed by a DAO -- such as creating, updating, or deleting data -- is associated with a transaction. As such, the concept of *transaction demarcation* is extremely important.

Transaction demarcation is the manner in which transaction boundaries are defined. The J2EE specification describes two models for transaction demarcation: programmatic and declarative. Table 1 breaks down the two models:

Table 1. Two models of transaction demarcation

Declarative transaction demarcation	Programmatic transaction demarcation
The programmer declares transaction attributes using an EJB deployment descriptor.	The programmer is responsible for coding transaction logic.
The run-time environment (the EJB container) uses the attributes to automatically manage transactions.	The application controls the transaction via an API.

We'll focus on programmatic transaction demarcation.

Design considerations
As stated previously, DAOs are transactional objects. A typical DAO performs transactional operations such as create, update, and delete. When designing a DAO, start by asking yourself the following questions:

- How will transactions start?
- How will transactions end?
- Which object will be responsible for starting a transaction?
- Which object will be responsible for ending a transaction?
- Should the DAO be responsible for starting and ending transactions?
- Will the application need to access data across multiple DAOs?
- Will a transaction involve one DAO or multiple DAOs?
- Will a DAO invoke methods on another DAO?

Knowing the answers to these questions will help you choose the transaction demarcation strategy that is best for your DAOs. There are two main strategies for transaction demarcation in DAOs. One approach makes the DAO responsible for demarcating transactions; the other defers transaction demarcation to the object that is calling the DAO's methods. If you choose the former approach, you will embed transaction code inside the DAO class. If you choose the latter approach, transaction demarcation code will be external to the DAO class. We'll use simple code examples to better understand how each of these approaches works.

Listing 1 shows a DAO with two data operations: create and update:

Listing 1. DAO methods

```
      public void createWarehouseProfile(WHProfile profile);
      public void updateWarehouseStatus(WHIdentifier id, StatusInfo status);
```

Listing 2 shows a simple transaction. The transaction demarcation code is external to the DAO class. Notice how the caller in this example combines multiple DAO operations within the transaction.

Listing 2. Caller-managed transaction

```
tx.begin();    // start the transaction
dao.createWarehouseProfile(profile);
dao.updateWarehouseStatus(id1, status1);
dao.updateWarehouseStatus(id2, status2);
tx.commit();   // end the transaction
```

This transaction demarcation strategy is especially valuable for applications that need to access multiple DAOs in a single transaction.

You can implement transaction demarcation using either the JDBC API or the Java Transaction API (JTA). JDBC transaction demarcation is simpler than JTA transaction demarcation, but JTA provides greater flexibility. In the sections that follow we'll take a closer look at the mechanics of transaction demarcation.

Transaction demarcation with JDBC

JDBC transactions are controlled using the `Connection` object. The JDBC Connection interface (`java.sql.Connection`) provides two transaction modes: auto-commit and manual commit. The `java.sql.Connection` offers the following methods for controlling transactions:

- `public void setAutoCommit(boolean)`
- `public boolean getAutoCommit()`
- `public void commit()`
- `public void rollback()`

Listing 3 shows how to demarcate a transaction using the JDBC API:

Listing 3. Transaction demarcation with the JDBC API

```
import java.sql.*;
import javax.sql.*;

// ...
DataSource ds = obtainDataSource();
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
// ...
pstmt = conn.prepareStatement("UPDATE MOVIES ...");
pstmt.setString(1, "The Great Escape");
pstmt.executeUpdate();
// ...
conn.commit();
// ...
```

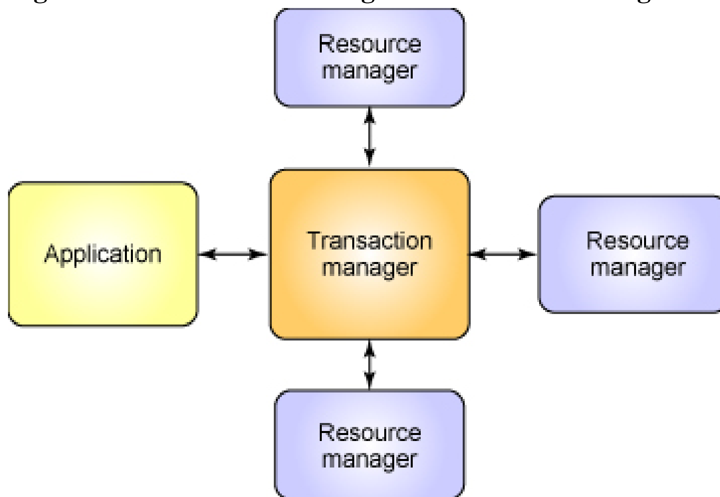
With JDBC transaction demarcation, you can combine multiple SQL statements into a single transaction. One of the drawbacks of JDBC transactions is that the transaction's scope is limited to a single database connection. A JDBC transaction cannot span multiple databases. Next, we'll see how transaction demarcation is done using JTA. Because JTA is not as widely known as JDBC, we'll start with an overview.

Overview of JTA

The Java Transaction API (JTA) and its sibling, the Java Transaction Service (JTS), provide distributed transaction services for the J2EE platform. A *distributed transaction* involves a transaction manager and one or more resource managers. A *resource manager* is any kind of persistent datastore. The transaction manager is responsible for coordinating communication between all transaction participants. The relationship between the transaction manager

and resource managers is shown in Figure 2:

Figure 2. A transaction manager and resource managers



JTA transactions are more powerful than JDBC transactions. While a JDBC transaction is limited to a single database connection, a JTA transaction can have multiple participants. Any one of the following Java platform components can participate in a JTA transaction:

- JDBC connections
- JDO `PersistenceManager` objects
- JMS queues
- JMS topics
- Enterprise JavaBeans
- A resource adapter that complies with the J2EE Connector Architecture specification

Transaction demarcation with JTA
To demarcate a transaction with JTA, the application invokes methods on the `javax.transaction.UserTransaction` interface. Listing 4 shows a typical JNDI lookup for the `UserTransaction` object:

Listing 4. A JNDI lookup for the UserTransaction object

```
import javax.transaction.*;
import javax.naming.*;
// ...
InitialContext ctx = new InitialContext();
Object txObj = ctx.lookup("java:comp/UserTransaction");
UserTransaction utx = (UserTransaction) txObj;
```

After the application has a reference to the `UserTransaction` object it may start the transaction, as shown in Listing 5:

Listing 5. Starting a transaction with JTA

```
utx.begin();
// ...
DataSource ds = obtainXADataSource();
Connection conn = ds.getConnection();
pstmt = conn.prepareStatement("UPDATE MOVIES ...");
pstmt.setString(1, "Spinal Tap");
pstmt.executeUpdate();
// ...
utx.commit();
// ...
```

When the application invokes `commit()`, the transaction manager uses a two-phase commit protocol to end the transaction.

JTA methods for transaction control
The `javax.transaction.UserTransaction` interface provides the following transaction control methods:

- `public void begin()`
- `public void commit()`
- `public void rollback()`
- `public int getStatus()`
- `public void setRollbackOnly()`
- `public void setTransactionTimeout(int)`

To start a transaction the application calls `begin()`. To end a transaction the application calls either `commit()` or `rollback()`. See Resources to learn more about transaction management with JTA.

Using JTA and JDBC
Developers often use JDBC for low-level data operations in DAO classes. If you plan to demarcate transactions with JTA, you will need a JDBC driver that implements the `javax.sql.XADataSource`, `javax.sql.XAConnection`, and `javax.sql.XAResource` interfaces. A driver that implements these interfaces will be able to participate in JTA transactions. An `XADataSource` object is a factory for `XAConnection` objects. `XAConnections` are JDBC connections that participate in JTA transactions.

You will be required to set up the `XADataSource` using your application server's administrative tools. Consult the application server documentation and the JDBC driver documentation for specific instructions.

J2EE applications look up the data source using JNDI. Once the application has a reference to the data source object, it will call `javax.sql.DataSource.getConnection()` to obtain a connection to the database.

XA connections are different from non-XA connections. Always remember that XA connections are participating in a JTA transaction. This means that XA connections do not support JDBC's auto-commit feature. Also, the application must not invoke `java.sql.Connection.commit()` or `java.sql.Connection.rollback()` on an XA connection. Instead, the application should use `UserTransaction.begin()`, `UserTransaction.commit()`, and `UserTransaction.rollback()`.

Choosing the best approach
We've discussed how to demarcate transactions with both JDBC and JTA. Each approach has its advantages and you will need to decide which one is most appropriate for your application.

On many recent projects our team has built DAO classes using the JDBC API for transaction demarcation. These DAO classes can be summarized as follows:

- Transaction demarcation code is embedded inside the DAO class.
- The DAO class uses the JDBC API for transaction demarcation.

- The caller has no way to demarcate the transaction.
- Transaction scope is limited to a single JDBC Connection.

JDBC transactions are not always suitable for complex enterprise applications. If your transactions will span multiple DAOs or multiple databases the following implementation strategy may be more appropriate:

- Transactions are demarcated with JTA.
- Transaction demarcation code is separated from the DAO.
- The caller is responsible for demarcating the transaction.
- The DAO participates in a global transaction.

The JDBC approach is attractive due to its simplicity; the JTA approach offers greater flexibility. The implementation you choose will depend on the specific needs of your application.

Logging and DAOs
A well-implemented DAO class will use logging to capture details about its run-time behavior. You may choose to log exceptions, configuration information, connection status, JDBC driver metadata, or query parameters. Logs are useful in all phases of development. I often examine application logs during development, during testing, and in production.

In this section, I'll present a code example that shows how to incorporate Jakarta Commons Logging into a DAO. Before we get to that, let's review a couple of basics.

Choosing a logging library
Many developers use a primitive form of logging: `System.out.println` and `System.err.println`. `Println` statements are quick and convenient but they do not offer the power of a full-featured logging system. Table 2 lists logging libraries for the Java platform:

Table 2. Logging libraries for the Java platform

Logging library	Open source?	URL
java.util.logging	No	http://java.sun.com/j2se/
Jakarta Log4j	Yes	http://jakarta.apache.org/log4j/
Jakarta Commons Logging	Yes	http://jakarta.apache.org/commons/logging.html

`java.util.logging` is the standard API for the J2SE 1.4 platform. Most developers would agree, however, that Jakarta Log4j delivers greater functionality and more flexibility. One of the advantages of Log4j over java.util.logging is that it supports both the J2SE 1.3 and J2SE 1.4 platforms.

Jakarta Commons Logging can be used in conjunction with `java.util.logging` or Jakarta Log4j. Commons Logging is a logging abstraction layer that isolates your application from the underlying logging implementation. With Commons Logging, you can swap the underlying logging implementation by changing a configuration file. Commons Logging is used in Jakarta Struts 1.1 and Jakarta HttpClient 2.0.

A logging example
Listing 7 shows how to use Jakarta Commons Logging in a DAO class:

Listing 7. Jakarta Commons Logging in a DAO class

```
import org.apache.commons.logging.*;

class DocumentDAOImpl implements DocumentDAO
{
      static private final Log log = LogFactory.getLog(DocumentDAOImpl.class);

      public void deleteDocument(String id)
      {
          // ...
          log.debug("deleting document: " + id);
          // ...
          try
          {
              // ... data operations ...
          }
          catch (SomeException ex)
          {
              log.error("Unable to delete document", ex);
              // ... handle the exception ...
        }
      }
}
```

Logging is an essential part of any mission-critical application. If you encounter a failure in a DAO, logs often provide the best information for understanding what went wrong. Incorporating logging into your DAOs ensures you will be equipped for debugging and troubleshooting.

Exception handling in DAOs

We've looked at transaction demarcation and logging and you now have a deeper understanding of how each applies to data access objects. Our third and final discussion point is exception handling. Following a few simple exception handling guidelines will make your DAOs easier to use, more robust, and more maintainable.

When implementing the DAO pattern, consider the following questions:

- Will methods in the DAO's public interface throw checked exceptions?
- If yes, what checked exceptions will be thrown?
- How will exceptions be handled within the DAO implementation class?

In the process of working with the DAO pattern, our team developed a set of guidelines for exception handling. Follow these guidelines to greatly improve your DAOs:

- DAO methods should throw meaningful exceptions.
- DAO methods should not throw `java.lang.Exception`. A `java.lang.Exception` is too generic. It does not convey any information about the underlying problem.
- DAO methods should not throw `java.sql.SQLException`. SQLException is a low-level JDBC exception. A DAO should strive to encapsulate JDBC rather than expose JDBC to the rest of the application.
- Methods in the DAO interface should throw checked exceptions only if the caller can reasonably be expected to handle the exception. If the caller won't be able to handle the exception in a meaningful way, consider throwing an unchecked (run-time) exception.
- If your data access code catches an exception, do not ignore it. DAOs that ignore caught exceptions are difficult to troubleshoot.

- Use chained exceptions to translate low-level exceptions into high-level ones.
- Consider defining standard DAO exception classes. The Spring Framework (see Resources) provides an excellent set of predefined DAO exception classes.

See Resources for more detailed information about exceptions and exception handling techniques.

Implementation example: MovieDAO
`MovieDAO` is a DAO that demonstrates all of the techniques discussed in this article: transaction demarcation, logging, and exception handling. You will find the `MovieDAO` source in the Resources section. The code is divided into three packages:

- `daoexamples.exception`
- `daoexamples.movie`
- `daoexamples.moviedemo`

This implementation of the DAO pattern consists of the classes and interfaces shown below:

- `daoexamples.movie.MovieDAOFactory`
- `daoexamples.movie.MovieDAO`
- `daoexamples.movie.MovieDAOImpl`
- `daoexamples.movie.MovieDAOImplJTA`
- `daoexamples.movie.Movie`
- `daoexamples.movie.MovieImpl`
- `daoexamples.movie.MovieNotFoundException`
- `daoexamples.movie.MovieUtil`

The `MovieDAO` interface defines the DAO's data operations. The interface has five methods, as shown here:

- `public Movie findMovieById(String id)`
- `public java.util.Collection findMoviesByYear(String year)`
- `public void deleteMovie(String id)`
- `public Movie createMovie(String rating, String year, String, title)`
- `public void updateMovie(String id, String rating, String year, String title)`

The `daoexamples.movie` package contains two implementations of the `MovieDAO` interface. Each implementation uses a different approach to transaction demarcation, as shown in Table 3:

Table 3. MovieDAO implementations

	MovieDAOImpl	**MovieDAOImplJTA**
Implements the MovieDAO interface?	Yes	Yes
Obtains DataSource via JNDI?	Yes	Yes
Obtains java.sql.Connection objects from a DataSource?	Yes	Yes
DAO demarcates transactions internally?	Yes	No
Uses JDBC transactions?	Yes	No
Uses an XA DataSource?	No	Yes
Participates in JTA transactions?	No	Yes

MovieDAO demo application
The demo application is a servlet class called `daoexamples.moviedemo.DemoServlet`. `DemoServlet` uses both of the Movie DAOs to query and update movie data in a table.

The servlet demonstrates how to combine the JTA-aware `MovieDAO` and the Java Message Service in a single transaction, as shown in Listing 8.

Listing 8. Combining MovieDAO and JMS code in a single transaction

```
        UserTransaction utx = MovieUtil.getUserTransaction();
        utx.begin();
        batman = dao.createMovie("R",
                        "2008",
                        "Batman Reloaded");
        publisher = new MessagePublisher();
        publisher.publishTextMessage("I'll be back");
        dao.updateMovie(topgun.getId(),
                        "PG-13",
                        topgun.getReleaseYear(),
                        topgun.getTitle());
        dao.deleteMovie(legallyblonde.getId());
        utx.commit();
```

To run the demo application, configure an XA datasource and a non-XA datasource in your application server. Then, deploy the daoexamples.ear file. The application will run in any J2EE 1.3-compliant application server. See Resources to obtain the EAR file and source code.

Conclusion

As this article has shown, implementing the DAO pattern entails more than just writing low-level data access code. You can start building better DAOs today by choosing a transaction demarcation strategy that is appropriate for your application, by incorporating logging in your DAO classes, and by following a few simple guidelines for exception handling.

Resources

- Download the MovieDAO source code at daoexamples.sourceforge.net
- Want to learn more about the Data Access Object pattern? Start with the Core J2EE Patterns home page.
- Kyle Brown's "A stepped approach to J2EE testing with SDAO" (*developerWorks*, March 2003) provides a short introduction to data access objects and the DAO design pattern.
- The Dragonslayer tutorial "Create persistent application data with Java Data Objects" (*developerWorks*, July 2003) shows you how to combine Struts and the DAO pattern for low-impact enterprise data persistence.
- Srikanth Shenoy's "Best practices in EJB exception handling" (*developerWorks*, May 2002) introduces both exception handling basics and logging with Log4j.
- The *Java theory and practice series* offers a three-part introduction to the Java Transaction API, starting with "Understanding JTS -- An introduction to transactions" (*developerWorks*, March 2002).
- The Java Transaction API is a key part of the J2EE platform.
- Jakarta Log4j is a world-class logging library for Java applications.
- Jakarta Commons Logging provides an easy-to-use logging abstraction layer.
- The Spring Framework provides abstraction layers for JDBC and transaction management. Additionally, the

framework contains standardized DAO exception classes and JNDI helper classes.

- Rod Johnson's *J2EE Design and Development* (Wrox Press, 2002) belongs on every J2EE developer's bookshelf. The book is full of application design strategies, practical programming techniques, and real-world examples.

- Josh Bloch's *Effective Java Programming Language Guide* (Addison Wesley, 2001) presents best practices for exception handling and class library design.

- See the Java technology zone tutorials page for a complete listing of free Java technology tutorials from *developerWorks*.

- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks* Java technology zone.

About the author

Sean C. Sullivan is a software engineer working in Portland, Oregon. His most recent projects include building supply chain management applications and an Internet e-commerce payment system. He has also worked on operating system and CAD software projects at IBM and Image Systems Technology. Sean is an Apache Jakarta developer, having contributed code to the Jakarta HttpClient project. He has been developing applications with Java since 1996 and is the author of *Programming with the Java Media Framework*, published by John Wiley & Sons. Sean holds a BS in Computer Science from Rensselaer. He can be reached at dao-article@seansullivan.com.

What do you think of this document?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Comments?