HOW-TO

JAVAWORLD

ng Spring

he J2EE container.

By Murali Kosaraju

JavaWorld |

APRIL 27, 2007 01:00 AM PT

Using J(2)EE application server has been a norm when high-end features like transactions, security, availability, and scalability are mandatory. There are very few options for java applications, which require only a subset of these enterprise features and, more often than not, organizations go for a full-blown J(2)EE server. This article focuses on distributed transactions using the JTA (Java Transaction API) and will elaborate on how distributed transactions (also called XA) can be used in a standalone java application, without a JEE server, using the widely popular Spring framework and the open source JTA implementations of JBossTS, Atomikos and Bitronix.

Distributed transaction processing systems are designed to facilitate transactions that span heterogeneous, transaction-aware resources in a distributed environment. Using distributed transactions, an application can accomplish tasks such as retrieving a message from a message queue and updating one or more databases in a single transactional unit adhering to the ACID (Atomicity, Consistency, Isolation and Durability) criteria. This article outlines some of the use cases where distributed transactions (XA) could be used and how an application can achieve transactional processing using JTA along with the best of the breed technologies. The main focus is on using Spring as a server framework and how one can integrate various JTA implementations seamlessly for enterprise level distributed transactions.

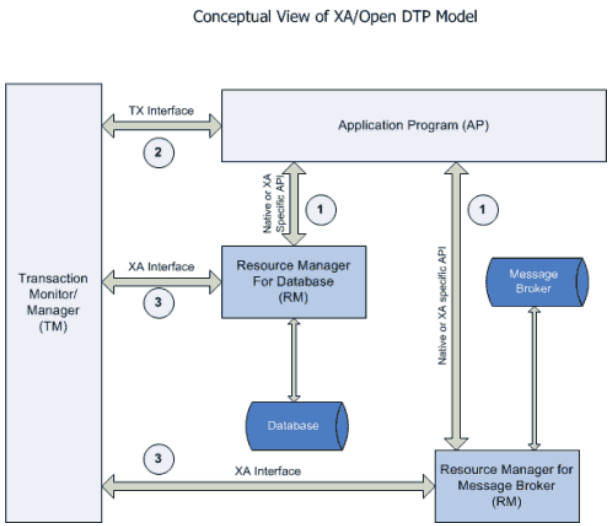
XA Transactions and the JTA API

Since the scope of the article is limited to using JTA implementations using the Spring framework, we will briefly touch upon the architectural concepts of distributed transaction processing.

JAVAWORLD

The *X/Open* Distributed Transaction Processing, designed by Open Group(a vendor consortium), defines a standard communication architecture that allows multiple applications to share resources provided by multiple resource managers, and allows their work to be coordinated into global transactions. The *XA* interfaces enable the resource managers to join transactions, to perform *2PC* (two phase commit) and to recover in-doubt transactions following a failure.

Figure 1: Conceptual model of the DTP environment.



As shown in Figure 1, the model has the following interfaces:

1. The interface between the *application and the resource manager* allows an application to call a resource manager directly, using the resource manager's native API or native XA API depending on if the transaction needs to be managed by the transaction monitor or not.

2. The interface between the *application and the transaction monitor* (TX interface), lets the application call the transaction monitor for all transaction needs like starting a transaction, ending a transaction, rollback of a transaction etc.

3. The interface between the *transaction monitor and the resource manager* is the XA interface. This is the interface, which facilitates the two phase commit protocol to achieve distributed transactions under one global transaction.

JTA API

The *JTA API*, defined by Sun Microsystems, is a high-level API which defines interfaces between a transaction manager and the parties involved in a distributed transaction system. The JTA primarily consists of three parts:

- A high-level application interface for an application to demarcate transaction boundaries. The `UserTransaction` interface encapsulates this.
- A Java mapping of the industry standard X/Open XA protocol (Item #3 in the X/Open interfaces listed above). This encompasses the interfaces defined in the `javax.transaction.xa` package, which consists of `XAResource, Xid and XAException` interfaces.
- A high-level transaction manager interface that allows an application server to manage transactions for a user application. The `TransactionManager, Transaction, Status and Synchronization` interfaces pretty much define how the application server manages transactions.

Now that we have a brief summary of what JTA and XA standards are, let us go through some use cases to demonstrate the integration of different JTA implementations using Spring, for our hypothetical java application.

Our Use Cases

To demonstrate the integration of different JTA implementations with Spring, we are going to use the following use cases:

1. *Update two database resources in a global transaction*- We will use JBossTS as the JTA implementation. In the process, we will see how we can declaratively apply distributed transaction semantics to simple POJO's.
2. *Update a database and send a JMS message to a queue in a global transaction*- We will demonstrate integration with both Atomikos and Bitronix JTA implementations.
3. *Consume a JMS message and update a database in a global transaction*- We will use both Atomikos and Bitronix JTA implementations. In the process, we will see how we can emulate transactional MDP's (Message Driven POJO's).

We will be using MySQL for the databases and Apache ActiveMQ as our JMS messaging provider for our use cases. Before going through the use cases, let us briefly look at the technology stack we are going to use.

Spring framework

Spring framework has established itself as one of the most useful and productive frameworks in the Java world. Among the many benefits it provides, it also provides the necessary plumbing for running an application with any JTA implementation. This makes it unique in the sense that an application doesn't need to run in a JEE container to get the benefits of JTA

transactions. Please note that Spring doesn't provide any JTA implementation as such. The only task from the user perspective is to make sure that the JTA implementation is wired to use the Spring framework's JTA support. This is what we will be doing in the following sections.

Transactions in Spring

Spring provides both programmatic and declarative transaction management using a lightweight transaction framework. This makes it easy for standalone java applications to include transactions (JTA or non-JTA) either programmatically or declaratively. The programmatic transaction demarcation can be accomplished by using the API exposed by the `PlatformTransactionManager` interface and its sub-classes. On the other hand, the declarative transaction demarcation uses an AOP (Aspect Oriented Programming) based solution. For this article, we will explore the declarative transaction demarcation, since it is less intrusive and easy to understand, using the `TransactionProxyFactoryBean` class. The transaction management strategy, in our case, is to use the `JtaTransactionManager`, since we have multiple resources to deal with. If there is only a single resource, there are several choices depending on the underlying technology and all of them implement the `PlatformTransactionManager` interface. For example, for Hibernate, one can choose to use `HibernateTransactionManager` and for JDO based persistence, one can use the `JdoTransactionManager`. There is also a `JmsTransactionManager`, which is meant for local transactions only.

Spring's transaction framework also provides the necessary tools for applications to define the transaction propagation behavior, transaction isolation and so forth. For declarative transaction management, the `TransactionDefinition` interface specifies the propagation behavior, which is very much similar to EJB CMT attributes. The `TransactionAttribute` interface allows the application to specify which exceptions will cause a rollback and which ones will be committed. These are the two crucial interfaces, which make the declarative transaction management very easy to use and configure, and we will see as we go through our use cases.

Asynchronous Message Consumption using Spring

Spring has always supported sending messages using JMS API via its JMS abstraction layer. It employs a callback mechanism, which consists of a message creator and a JMS template that actually sends the message created by the message creator.

.0, asynchronous message consumption has been made possible using the JMS API. Though Spring provides different message listener containers, for consuming the messages, the one that is mostly suited to both JEE and J2SE environments is the `DefaultMessageListenerContainer` (DMLC). The `DefaultMessageListenerContainer` extends the `AbstractPollingMessageListenerContainer` class and provides full support for JMS 1.1 API. It primarily uses the JMS synchronous receive calls(`MessageConsumer.receive()`) inside a loop and allows for transactional reception of messages. For J(2)SE environment, the stand-alone JTA implementations can be wired to use the Spring's `JtaTransactionManager`, which will be demonstrated in the following sections.

The JTA implementations

JBossTS

JBossTS, formerly known as Arjuna Transaction Service, comes with a very robust implementation, which supports both JTA and JTS API. JBossTS comes with a recovery service, which could be run as a separate process from your application processes. Unfortunately, it doesn't support out-of-the box integration with Spring, but it is easy to integrate as we will see in our exercise. Also there is no support for JMS resources, only database resources are supported.

Atomikos Transaction Essentials

Atomikos's JTA implementation has been open sourced very recently. The documentation and literature on the internet shows that it is a production quality implementation, which also supports recovery and some exotic features beyond the JTA API. Atomikos provides out of the box Spring integration along with some nice examples. Atomikos supports both database and JMS resources. It also provides support for pooled connections for both database and JMS resources.

Bitronix JTA

JAVAWORLD

n is fairly new and is still in beta. It also claims to support transaction recovery as good as or even mercial products. Bitronix provides support for both database and JMS resources. Bitronix also provides connection pooling and session pooling out of the box.

XA Resources

JMS Resources

The JMS API specification does not require that a provider supports distributed transactions, but if the provider does, it should be done via the JTA XAResource API. So the provider should expose its JTA support using the `XAConnectionFactory`, `XAConnection` and `XASession` interfaces. Fortunately Apache's ActiveMQ provides the necessary implementation for handling XA transactions. Our project (see Resources section) also includes configuration files for using TIBCO EMS (JMS server from TIBCO) and one can notice that the configuration files require minimal changes when the providers are switched.

Database Resources

MySQL database provides an XA implementation and works only for their InnoDB engines. It also provides a decent JDBC driver, which supports the JTA/XA protocol. Though there are some restrictions placed on the usage of some XA features, for the purposes of the article, it is good enough.

The Environment

Setup for Databases:

The first database **mydb1** will be used for use cases 1 and 2 and will have the following table:

```
mysql> use mydb1;
Database changed
mysql> select * from msgseq;
+---------+-----------+-------+
| APPNAME | APPKEY    | VALUE |
+---------+-----------+-------+
| spring  | execution |    13 |
+---------+-----------+-------+
1 row in set (0.00 sec)
```

The second database **mydb2** will be used for use case 3 and will have the following table:

```
mysql> use mydb2;
Database changed
mysql> select * from msgseq;
+---------+------------+-------+
| APPNAME | APPKEY     | VALUE |
+---------+------------+-------+
| spring  | aaaaa      | 15    |
| spring  | allocation | 13    |
+---------+------------+-------+
2 rows in set (0.00 sec)
```

Setup for JMS provider (for use case 2 and 3)

For creating a physical destination in ActiveMQ, do the following:

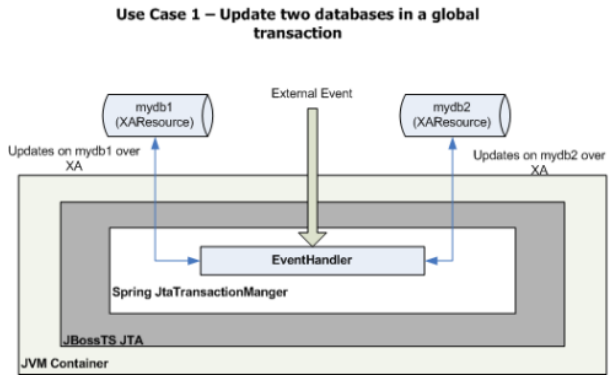
1. Add the following destination to the *activmq.xml* file under the *conf* folder of ActiveMQ installation:

```
<destinations>
.calName="test.q1" />
;>
```

2. Add the following line of code in the *jndi.properties* file to include the jndi name for the destination and make sure the file is in the classpath: `queue.test.q1=test.q1`

Use Case1 - Updating two databases in a global transaction using JBossTS

Figure 2: UseCase1 updates two databases in a global transaction.



Let us assume that our application has a requirement where it needs to persist a sequence number, associated with an event, in two different databases(*mydb1* and *mydb2*), within the same transactional unit of work as shown in Figure 2 above. To achieve this, let us write a simple method in our POJO class, which updates the two databases.

The code for our `EventHandler` POJO class looks as follows:

JAVAWORLD