HOW-TO

JAVAWORLD

ng Spring

he J2EE container.

By Murali Kosaraju

JavaWorld |

APRIL 27, 2007 01:00 AM PT

Atomikos provides a generic wrapper class, which makes it easy to pass in the *xaDataSourceClassName*, which, in our case, is `com.mysql.jdbc.jdbc2.optional.MysqlXADataSource`. Same is the case for the JDBC url. The *exclusiveConnectionMode* is set to "true" to make sure that the connection in the current transaction is not shared. Atomikos provides connection pooling out of the box, and one can set the pool size using the *connectionPoolSize* attribute.

Let us now look at the relevant bean definitions for **Bitronix**:

```
<bean id="ConnectionFactory" factory-bean="ConnectionFactoryBean" factory-method="createResource" />
<bean id="dataSourceBean1" class="bitronix.tm.resource.jdbc.DataSourceBean">
                    ' value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource" />
                    e" value="mysql" />
                     value="2" />
 <property name="driverProperties">
  <props>
   <prop key="user">root</prop>
   <prop key="password">murali</prop>
   <prop key="databaseName">mydb1</prop>
  </props>
 </property>
</bean>

<bean id="Db1DataSource" factory-bean="dataSourceBean1" factory-method="createResource" />
<bean id="BitronixTransactionManager" factory-method="getTransactionManager"
    class="bitronix.tm.TransactionManagerServices" depends-on="btmConfig,ConnectionFactory" destroy-method="shutdown" />
<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">
 <property name="transactionManager" ref="BitronixTransactionManager" />
 <property name="userTransaction" ref="BitronixTransactionManager" />
</bean>

<bean id="btmConfig" factory-method="getConfiguration" class="bitronix.tm.TransactionManagerServices">
    <property name="serverId" value="spring-btm-sender" />
</bean>
```

The *appSenderTemplate* bean defined for Atomikos can be re-used, with the only exception of the *sessionTransacted* value. This is set to "false", the default for the `JmsTemplate` in Spring anyway, so one can even ignore this attribute.

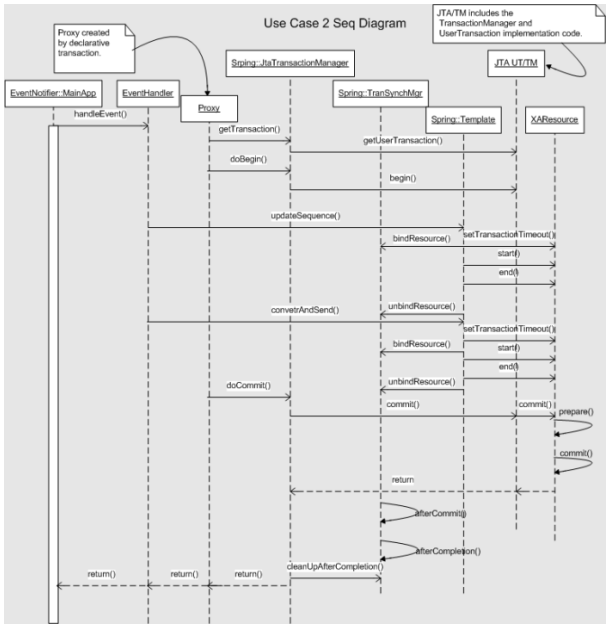
The datasource bean definition, shown above, looks similar to Atomikos, but the main difference is that the bean creation is done using an instance factory method rather than the static factory method. In this case, the *Db1DataSource* bean is created using the factory-bean *dataSourceBean1*. The factory method specified was *createResource*.

The *transactionManager* bean refers to `BitonixTransactionManager` for both the *transactionManager* attribute and the *userTransaction* attribute.

Look at the *AtomikosSender* task and the *BitronixSender* in the ant build file, provided as part of the project download.

The sequence diagram for this use case, which is by no means a comprehensive one, is shown below:

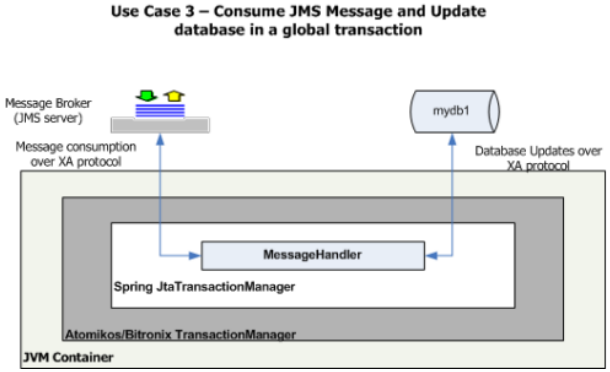
Figure 5: Sequence diagram, which illustrates the process flow for use case 2.



UseCase3 - (Transactional MDP's) Consume message and update database in a global transaction

Figure 4: UseCase3 consumes a JMS message and updates a database in a global transaction.

JAVAWORLD



This use case is different from the previous use cases and all it does is define a POJO to handle the messages received from a messaging provider. It also updates the database within the same transaction as shown in Figure 4 above.

The relevant code for our `MessageHandler` class looks as follows:

```
public void handleOrder(String msg) {
 log.debug("Receieved message->: " + msg);
 MessageSequenceDAO dao = (MessageSequenceDAO) MainApp.springCtx.getBean("sequenceDAO");
 String app = "spring";
 String appKey = "allocation";
 int upCnt = dao.updateSequence(value++, app, appKey);
 log.debug("Update SUCCESS!! Val: " + value + " updateCnt->"+ upCnt);
 if (fail)
  throw new RuntimeException("Rollback TESTING!!");
}
```

As you can see, the code just updates the database *mydb2*. The `MessageHandler` is just a POJO, which has a `handleOrder()` method. Using Spring we are going to transform this into a message driven POJO (analogous to MDB in a JEE server). To accomplish this we will use the `MessageListenerAdapter` class, which delegates the message handling to the target listening methods via reflection. This is a very convenient feature, which enables simple POJO's to be converted to message driven POJO's (beans). Our MDP now supports distributed transactions.

It's time for us to look at the configuration for more clarity:

```
<bean id="msgListener" class="com.findonnet.messaging.MessageHandler"/>
<bean id="msgHandler" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <property name="delegate" ref="msgHandler"/>
    <property name="defaultListenerMethod" value="handleOrder"/>
</bean>
```

The above configuration shows that the *msgListener* bean delegates the calls to the bean defined by the *msgHandler*. Also, we have specified the *handleOrder()*, which should be invoked when the message arrives from the message provider. This was done using the *defaultListenerMethod* attribute. Let us now look at the message listener, which listens to the destination on the message provider:

```
<bean id="listenerContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer"
destroy-method="close">
  <property name="concurrentConsumers" value="1"/>
  <property name="connectionFactory" ref="queueConnectionFactoryBean" />
  <property name="destination" ref="appJmsDestination"/>
  <property name="messageListener" ref="msgListener"/>
  <property name="transactionManager" ref="transactionManager"/>
  <property name="sessionTransacted" value="false"/>
  <property name="receiveTimeout" value="5000"/>
  <property name="recoveryInterval" value="6000"/>
  <property name="autoStartup" value="false"/>
</bean>
```

The *listenerContainer* in this case uses the `DefaultMessageListenerContainer` provided by Spring. The *concurrentConsumers* attribute is set to "1", which indicates that we will only have one consumer on the queue. This attribute is mainly used for draining the queues concurrently by spawning multiple listeners (threads), and is very useful in situations where you have a fast producer and slow consumer and the ordering of the messages is not important. With Spring 2.0.3 and

above there is support for dynamically adjusting the number of listeners based on load using the *maxConcurrentConsumers* attribute. The *recoveryInterval* attribute is used for recovery purposes and is useful when a messaging provider is down and we ringing the application down. This feature, however, runs in an infinite loop and keeps re-trying is running, which you may not want. Also, one has to be careful in properly disposing the DMLC, since there are background threads, which might still be trying to receive messages from the message provider even after the JVM is shutdown. As of Spring 2.0.4, this issue has been fixed. As mentioned before, the *sessionTransacted* attribute should be set to "true" for Atomikos only. The same bean definition for *listenerContainer* applies to both Atomikos and Bitronix.

Please note that the *transactionManger* attribute points to the bean definitions that were defined above (for usecase2) and we are just re-using the same bean definitions.

That's all there is to it, we just implemented our MDP which receives the message and updates the database in a single global transaction.

To run this use case, please look at the *AtomikosConsumer* task and the *BitronixConsumer* in the ant build file, provided as part of the project download.

Some AspectJ

To intercept the calls, between the Spring framework and the JTA code, an Interceptor class has been used and weaved into the runtime libraries of the JTA implementation jar files. It uses AspectJ and the code snippet is shown below:

```
pointcut xaCalls() : call(*  XAResource.*(..))
                  || call(*  TransactionManager.*(..))
                  || call(*  UserTransaction.*(..))

Object around() : xaCalls() {
 log.debug("XA CALL -> This: " + thisJoinPoint.getThis());
 log.debug("       -> Target: " + thisJoinPoint.getTarget());
 log.debug("       -> Signature: " + thisJoinPoint.getSignature());
 Object[] args = thisJoinPoint.getArgs();
 StringBuffer str = new StringBuffer(" ");
 for(int i=0; i< args.length; i++) {
  str.append(" [" + i + "] = " + args[i]);
 }
 log.debug(str);
 Object obj = proceed();
 log.debug("XA CALL RETURNS-> " + obj);
 return obj;
}
```

The above code defines a pointcut on all calls made to any JTA related code and it also defines an *around* advice, which logs the arguments being passed and the method return values. This will come in handy when we are trying to trace and debug issues with JTA implementations. The ant *build.xml* file in the project (see Resources) contains tasks to weave the aspect against the JTA implementations. Another option is to use the **MaintainJ** plugin for eclipse, which provides the same from the comfort of an IDE (Eclipse) and even generates the sequence diagram for the process flow.

Some Gotchas

Distributed transactions is a very complex topic and one should look out for implementations where transaction recovery is robust enough and provides all the ACID (Atomicity, Consistency, Isolation and Durability) criteria that the user or application expects. What we tested, in this article, was for pre-2PC exceptions (*remember the RuntimeException we were throwing to test*

llbacks?). Applications should thoroughly test JTA implementations for failures during the 2 phase commit process as they are the most crucial and troublesome.

JAVAWORLD we looked at provide recovery test cases, which make it easy to run against the implementation itself, and on the participating XA resources as well. Please note that using XA may turn out to be a huge performance concern especially when the transaction volumes are large. One should also look at support for 2PC optimizations like the "last resource commit", which might fit some application needs where only one of the participating resource cannot or need not support 2PC. Care should be taken about the XA features supported and restrictions imposed, if any, by the vendors of the database or the message provider. For example, MySQL doesn't support `suspend()`and `resume()` operations and also seems to have some restrictions on using XA and in some situations might even keep the data in an in-consistent state. To learn more about XA, *Principles of Transaction Processing* is a very good book, which covers the 2PC failure conditions and optimizations in great detail. Also, Mike Spille's blog (see Resources section) is another good resource, which focuses on XA within the JTA context and provides wealth of information, especially on failures during 2PC and helps understand more about XA transactions.

When using Spring framework for sending and receiving JMS messages, one should be wary of using the `JmsTemplate` and the `DefualtMessageListenerContainer` when running in a non-J(2)EE environment. In case of `JmsTemplate`, for every message that is sent there will be a new JMS connection created. Same is the case when using the `DefaultMessageListenerContainer` when receiving messages. Creating a heavy weight JMS connection is very resource-intensive and the application may not scale well under heavy loads. One option is to look for some sort of connection/session pooling support either from the JMS providers or third-party libraries. Another option is to use the `SingleConnnectionFactory` from Spring, which makes it easy to configure a single connection, which can be re-used. The sessions are, however created for every message being sent, and this may not be a real overhead since JMS sessions are lightweight. Same is the case when messages are being received irrespective of if they are transactional or not.

Conclusion

In this article we saw how Spring framework can be integrated with JTA implementations to provide distributed transactions and how it could cater to the needs of an application which required distributed transactions without the need for a full-blown

ıow-cased how Spring framework provided POJO based solutions and declarative transaction ıtrusion while promoting best design practices. The use cases we saw, also demonstrated how Spring provides us with a rich transaction management abstraction, enabling us to easily switch between different JTA providers seamlessly.

Author bio

Murali Kosaraju works as a technical architect at Wachovia Bank in Charlotte, North Carolina. He holds a masters degree in Systems and Information engineering and his interests include messaging, service oriented architectures (SOA), Web Services, JEE and .NET centric applications. He currently lives with his wife Vidya and son Vishal in South Carolina.

Learn more about this topic

Download the source code (http://images.techhive.com/downloads/idge/imported/article/jvw/2007/04/xaspring.zip) (8M Zip file).

X/Open XA (http://www.opengroup.org/onlinepubs/009680699/toc.pdf)

JTA API (http://java.sun.com/products/jta/)

Spring Framework (http://www.springframework.org/)

ActiveMQ (http://activemq.apache.org/)

Mike Spille's Blog (http://jroller.com/page/pyrasun?catname=%2FXA)

MySQL XA issues (http://dev.mysql.com/doc/refman/5.0/en/xa-restrictions.html)

JTA Implementations:

1. JBossTS (http://www.jboss.com/products/transactions)

2. Atomikos (http://www.atomikos.com/)

3. Bitronix (http://www.bitronix.be/Btm/Overview)

JAVAWORLD

m)

(http://books.google.com/books?id=G-e7tvWJxZoC&dq=principles+of+transaction+processing&pg=PP1)

Follow everything from JavaWorld