JAVAWORLD

ng Spring

he J2EE container.

By Murali Kosaraju

JavaWorld |

APRIL 27, 2007 01:00 AM PT

Page 2 of 3

```
    public void handleEvent(boolean fail) throws Exception {
      MessageSequenceDAO dao = (MessageSequenceDAO) springContext.getBean("sequenceDAO");

                                                    tion";
      int upCnt = dao.updateSequence(value, app, appKey);
      log.debug(" sql updCnt->" + upCnt);

      if (springContext.containsBean("sequenceDAO2")) {
       // this is for use case 1 with JBossTS
       MessageSequenceDAO dao2 = (MessageSequenceDAO) springContext.getBean("sequenceDAO2");
       appKey = "allocation";
       upCnt = dao2.updateSequence(value, app, appKey);
       log.debug(" sql updCnt2->" + upCnt);
      }

         ...

      if (fail) {
       throw new RuntimeException("Simulating Rollback by throwing Exception !!");
      }
     }
```

As you can figure out, all we are doing in the first segment of the code is getting a reference to the `MessageSequenceDAO` object representing the first database and updating the value of the sequence.

As you can guess, the next piece of code updates the sequence in the second database.

The last `if` statement throws a run-time exception when we run the code with the boolean value set to "true". This is for simulating a run-time exception to test if the transaction manager has successfully rolled back the global transaction.

Let us look at the spring configuration file to see how we configured our DAO classes and the datasources:

```
<bean id="dsProps" class="java.util.Properties">
    <constructor-arg>
                                    ot</prop>
                                    '>murali</prop>
        <prop key="DYNAMIC_CLASS">com.findonnet.service.transaction.jboss.jdbc.Mysql</prop>
      </props>
    </constructor-arg>
</bean>

<bean id="dataSource1" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
 <property name="driverClassName">
  <value>com.arjuna.ats.jdbc.TransactionalDriver</value>
 </property>
 <property name="url" value="jdbc:arjuna:mysql://127.0.0.1:3306/mydb1"/>
 <property name="connectionProperties">
   <ref bean="dsProps"/>
 </property>
</bean>

<bean id="dataSource2" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
 <property name="driverClassName">
  <value>com.arjuna.ats.jdbc.TransactionalDriver</value>
 </property>
 <property name="url" value="jdbc:arjuna:mysql://127.0.0.1:3306/mydb2"/>
 <property name="connectionProperties">
   <ref bean="dsProps"/>
 </property>
</bean>

<bean id="sequenceDAO" class="com.findonnet.persistence.MessageSequenceDAO">
 <property name="dataSource">
  <ref bean="dataSource1"/>
 </property>
</bean>
```

```
<bean id="sequenceDAO2" class="com.findonnet.persistence.MessageSequenceDAO">
  <property name="dataSource">
```


The above beans define the two datasources for the two databases. To use JBossTS's TransactionalDriver, we need to register the database with either the JNDI bindings or with *Dynamic* class instantiations. We will be using the later, which requires us to implement the `DynamicClass` interface. The bean definition for *dsProps* shows that we are going to use the `com.findonnet.service.transaction.jboss.jdbc.Mysql` class, that we wrote, which implements the `DynamicClass` interface. Since JBossTS doesn't provide any out of the box wrappers for the MySQL database, we had to do this. In addition to this, the code relies on the jdbc URL starting with *"jdbc:arjuna"*, otherwise the JBossTS code throws errors.

The implementation of the `DynamicClass` interface is very simple, all we have to do is to implement the methods `getDataSource()` and the `shutdownDataSource()` methods. The `getDataSource()` method returns an appropriate `XADataSource` object, which, in our case is the `com.mysql.jdbc.jdbc2.optional.MysqlXADataSource.`

Please note that both the datasources are configured to use the `DriverManagerDataSource`, from Spring, which doesn't use any connection pooling and is not recommended for production use. The alternative is to use a pooled datasource, which can handle XA datasource pooling.

It's now time for us to look at providing transactional semantics to our `handleEvent()` method in our `EventHandler` class:

```
<bean id="eventHandlerTarget" class="com.findonnet.messaging.EventHandler"></bean>
<bean id="eventHandler" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
                    ionManager"><ref bean="transactionManager" /></property>
                    <ref bean="eventHandlerTarget"/></property>
                    ionAttributes">
    <props>
     <prop key="handle*">PROPAGATION_REQUIRED,-Exception</prop>
    </props>
  </property>
</bean>
```

Here we define the `TransactionProxyFactoryBean` and pass in the *transactionManager* reference, which we will see later, with the transaction attributes `"PROPAGATION_REQUIRED,-Exception"`. This will be applied to the target bean *eventHandlerTarget* and only on methods, which start with "handle" *(notice the handle*)*. To put it in simple words, what we are asking Spring framework to do is; for all method invocations on the target object, whose method names start with *"handle"*, please apply the transaction attributes `"PROPAGATION_REQUIRED,-Exception"`. Behind the scenes, the Spring framework will create a CGLIB based proxy, which intercepts all the calls on the `EventHandler` for the method names, that start with "handle". In case of any `Exception`, within the method call, the current transaction will be rolled back and that is what the "-Exception" means. This demonstrates how easy it is providing transactional support, declaratively, using Spring.

Now let us look at how we can wire up the Spring's `JtaTransactionManager` to use our choice of JTA implementation. The `eventHandler` bean defined above uses the *transactionManager* attribute, which will refer to the Spring `JtaTransactionManager` as shown below:

```
<bean id="jbossTransactionManager"
 class="com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple">
 <property name="transactionTimeout" value="1000" />
</bean>

<bean id="jbossUserTransaction"
 class="com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple"/>

<bean id="transactionManager"  class="org.springframework.transaction.jta.JtaTransactionManager">
 <property name="transactionManager">
  <ref bean="jbossTransactionManager" />
 </property>
 <property name="userTransaction">
  <ref bean="jbossUserTransaction" />
 </property>
</bean>
```

As shown above in the configuration, we are wiring up the spring provided `JtaTransactionManager` class to use `com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple` as the TransactionManager implementation and `com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple` as the UserTransaction implementation.

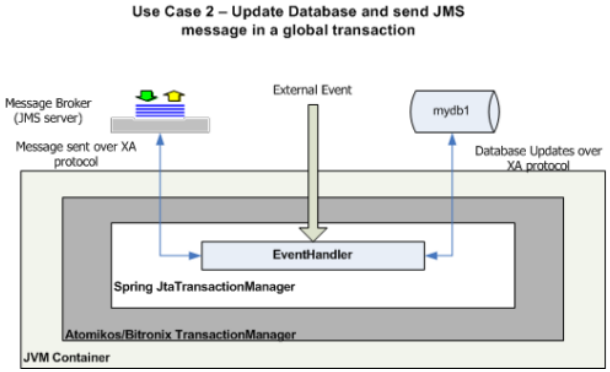
That's all there is to it. We have just enabled XA transactions using Spring and JbossTS with MySQL datasources acting as XA resources. Please note that at the time of writing there are some known issues with MySQL XA implementation (See the resources section).

To run this use case please look at the *JbossSender* task in the ant build file. The project download is provided in the Resources section.

Use Case2 - Update database and send JMS message in a global transaction

Figure 3: UseCase2 updates a database and sends a JMS message in a global transaction.

JAVAWORLD



This use case uses the same code we used for use case 1. The code updates the database *mydb1* and then sends a message to a message queue in a global transaction as shown in Figure 3 above. Both *Atomikos* and *Bitronix* configurations will be dealt with, in that order.

The POJO is same as the one we used for usecase1, the `EventHandler` class, and the relevant code looks as follows:

```
    public void handleEvent(boolean fail) throws Exception {

        = (MessageSequenceDAO) springContext.getBean("sequenceDAO");

        String appKey = "execution";
        int upCnt = dao.updateSequence(value, app, appKey);
        log.debug(" sql updCnt->" + upCnt);

           ...

        if (springContext.containsBean("appSenderTemplate")) {
         this.setJmsTemplate((JmsTemplate) springContext.getBean("appSenderTemplate"));
         this.getJmsTemplate().convertAndSend("Testing 123456");
         log.debug("Sent message succesfully");
        }
        if (fail) {
         throw new RuntimeException("Simulating Rollback by throwing Exception !!");
        }
    }
```

The first code snippet in the above mentioned code updates the *msgseq* table in the database *mydb1* with the sequence number.

The next piece of code uses the *appSenderTemplate*, which is configured to use the `JmsTemplate` class from Spring. This template is used to send JMS messages to the message provider. We will see how this is defined in the configuration file in the following segment.

An external event, in this case the `MainApp` class, will invoke the method `handleEvent` shown in Figure 3 above.

The last `if` statement is for simulating a run-time exception to test if the transaction manager has successfully rolled back the global transaction.

Let us look at the JNDI bean definitions for our messaging needs in the spring configuration file:


```
lass="org.springframework.jndi.JndiTemplate">
ıt">
   <prop key="java.naming.factory.initial">
    org.apache.activemq.jndi.ActiveMQInitialContextFactory
   </prop>
  </props>
 </property>
</bean>

<bean id="appJmsDestination"
 class="org.springframework.jndi.JndiObjectFactoryBean">
 <property name="jndiTemplate">
  <ref bean="jndiTemplate"/>
 </property>
 <property name="jndiName" value="test.q1"/>
</bean>

 <bean id="appSenderTemplate"
   class="org.springframework.jms.core.JmsTemplate">
 <property name="connectionFactory">
   <ref bean="queueConnectionFactoryBean"/>
 </property>
 <property name="defaultDestination">
   <ref bean="appJmsDestination"/>
 </property>
 <property name="messageTimestampEnabled" value="false"/>
 <property name="messageIdEnabled" value="false"/>
 <!-- sessionTransacted should be true only for Atomikos -->
 <property name="sessionTransacted" value="true"/>
</bean>
```

Here we are specifying a `JndiTemplate` for Spring to do a JNDI lookup on the destination specified by the `appJmsDestination` bean. The `appJmsDestination` bean has been wired with the `appSenderTemplate` (JmsTemplate) bean definitions also show that `appSenderTemplate` is wired to use the `Bean`, which we will see later. For Atomikos, the `sessionTransacted` property should be set to "true", which is not advised by the Spring framework and the literature on JMS seem to support that viewpoint. This is a **hack** we need to do only for Atomikos implementation. If this is set to "false", you will notice some Heuristic Exceptions thrown during the 2PC protocol. This is mainly attributed to the `prepare()` call not responding on the JMS resource, and eventually, Atomikos decides to rollback resulting in a Heuristic Exception. The *messageTimestampEnabled* and the *messageIdEnabled* attributes are set to "false" so that these are not generated since we are not going to use them anyway. This will reduce the overhead on the JMS provider and improves performance.

Let us look at the spring configuration beans for **Atomikos**:

```
<bean id="xaFactory" class="org.apache.activemq.ActiveMQXAConnectionFactory">
      <constructor-arg>
        <value>tcp://localhost:61616</value>
```

```
<bean id="queueConnectionFactoryBean"
   class="com.atomikos.jms.QueueConnectionFactoryBean" init-method="init">
   <property name="resourceName">
    <value>Execution_Q</value>
   </property>
   <property name="xaQueueConnectionFactory">
    <ref bean="xaFactory" />
   </property>
</bean>

<bean id="atomikosTransactionManager"
 class="com.atomikos.icatch.jta.UserTransactionManager" init-method="init" destroy-method="close">
 <property name="forceShutdown"><value>true</value></property>
</bean>

<bean id="atomikosUserTransaction" class="com.atomikos.icatch.jta.UserTransactionImp"/>

<bean id="transactionManager"
 class="org.springframework.transaction.jta.JtaTransactionManager">
 <property name="transactionManager">
  <ref bean="atomikosTransactionManager" />
 </property>
 <property name="userTransaction">
  <ref bean="atomikosUserTransaction" />
 </property>
</bean>
```

The `xaFactory` definition shows that the underlying xa connection factory being used is the `org.apache.activemq.ActiveMQXAConnectionFactory` class. The *transactionManager* bean is wired up to use the *atomikosTransactionManager* bean and the *atomikosUserTransaction* bean.

Let us now look at the datasource definition for **Atomikos**:

```
<bean id="dataSource" class="com.atomikos.jdbc.SimpleDataSourceBean" init-method="init"  destroy-method="close">
 <property name="uniqueResourceName"><value>Mysql</value></property>
 <property name="xaDataSourceClassName">
  <value>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</value>
 </property>
 <property name="xaDataSourceProperties">
  <value>URL=jdbc:mysql://127.0.0.1:3306/mydb1?user=root&password=murali</value>
 </property>
 <property name="exclusiveConnectionMode"><value>true</value></property>
</bean>
```