**Spring Transaction Management**

**What is a Transaction?**

Transaction is an activity or group of activities that are performed as a single unit of work. The characteristic of a transaction is that either all the activities that are part of the transaction are performed or none are performed. In other words, even if one of the activity fails then all other activities are cancelled and the system comes back to the state it was in when the transaction was started. This can be best explained by an example. Lets say you are planning on a vacation to a beautiful island and you pick up a nice looking resort. You call them up and book a room. To confirm your booking, you need to transfer money to their account. Transferring money from your account to their account is considered a transaction and these are the two activities that happen in that transaction.

* Money is debited from your account
* Money is credited to the resorts account

Both activities are required to make sure that the transaction is complete. If you money is withdrawn and some problem occurs in the system then the transaction is cancelled and the money is credited back to your account and you can try it all over again.

**Characteristics of a transaction (ACID)**

The software industry coined the acronym ACID for characteristics that a transaction must have. ACID stands for Atomic, Consistent, Isolated and Durable

* ***Atomic*** - Atomic says that either all activities of the transaction occur or none occur. i.e. even if one of the activity in the group fails that the other activities are cancelled (rolled back)
* ***Consistent*** - Once the transaction is complete the system is put back into a state that is properly defined. In our example, if the money were to be debited from your account and not credited to the resorts account then the system is not in consistent state since the money is now lost from the system. From a database point of view, consistent also means that all times none of the database constraints are violated. So even if you do not commit a transaction, at no point should a foreign key constraint or unique key constraint be violated
* ***Isolated*** *-*The transaction allows multiple people to work on the same data in a way that one transaction does not affect the data of the rest of the system. Therefore, two transactions can occur simultaneously without dirty reads. This is generally accomplished by locking the rows of the database or the database table
* ***Durable*** *-*The changes of the transaction are persisted to a permanent storage.

**Transaction Strategies**

Transaction strategies define how you want to implement transactions in your applications. There are mainly two ways to implement a transaction:

* Programmatic Implementation using Spring Transaction Managers and Transaction Templates
* Declarative Implementation in the Spring Application XML or using Annotations

**Spring Transaction Attributes**

**What are transaction attributes?**

Spring transactions allow setting up the propagation behavior, isolation, timeout and read only settings of a transaction. Before we delve into the details, here are some points that need to be kept in mind

* Isolation level and timeout settings get applied only after the transaction starts.
* Not all transaction managers specify all values and may throw exception with some non default values

**Propagation**

**PROPAGATION\_REQUIRED**This attribute tells that the code needs to be run in a transactional context. If a transaction already exists then the code will use it otherwise a new transaction is created. This is the default and mostly widely used transaction setting.

**PROPAGATION\_SUPPORTS**   
If a transaction exists then the code will use it, but the code does not require a new one. As an example, consider a ticket reservation system. A query to get total seats available can be executed non-transactionally. However, if used within a transaction context it will deduct tickets already selected and reduce them from the total count, and hence may give a better picture. This attribute should be used with care especially when PROPAGATION\_REQUIRED or PROPAGATION\_REQUIRES\_NEW is used within a PROPAGATION\_SUPPORTS context.

**PROPAGATION\_MANDATORY**   
Participates in an existing transaction, however if no transaction context is present then it throws a TransactionRequiredException

**PROPAGATION\_REQUIRES\_NEW**Creates a new transaction and if an existing transaction is present then it is suspended. In other words a new transaction is always started. When the new transaction is complete then the original transaction resumes. This transaction type is useful when a sub activity needs to be completed irrespective of the containing transaction. The best example of this is logging. Even if a transaction roll backs you still want to preserve the log statements. Transaction suspension may not work out of the box with all transaction managers, so make sure that the transaction manager supports transaction suspension

**PROPAGATION\_NOT\_SUPPORTED**This attribute says that transaction is not supported. In other words the activity needs to be performed non-transactionally. If an existing transaction is present then it is suspended till the activity finishes.

**PROPAGATION\_NEVER**This attributes says that the code cannot be invoked within a transaction. However, unlike PROPAGATION\_NOT\_SUPPORTED, if an existing transaction is present then an exception will be thrown

**PROPAGATION\_NESTED**   
The code is executed within a nested transaction if existing transaction is present, if no transaction is present then a new transaction is created. Nested transaction is supported out of the box on only certain transaction managers.

**Isolation Level**

Isolation is a property of a transaction that determines what effect a transaction has on other concurrent transactions. To completely isolate the transaction the database may apply locks to rows or tables. Before we go through the transaction levels, let us look at some problems that occur when transaction 1 reads data that is being modified by transaction 2.

* ***Dirty Reads***- Dirty reads occur when transaction 2 reads data that has been modified by transaction 1 but not committed. The problem occurs when transaction 1 rollbacks the transaction, in which case the data read by transaction 2 will be invalid.
* ***Non Repeatable Reads***- Nonrepeatable reads happen when a transaction fires the same query multiple times but receives different data each time for the same query. This may happen when another transaction has modified the rows while this query is in progress.
* ***Phantom Reads*** - Phantom reads occur when the collection of rows returned is different when a same query is executed multiple times in a transaction. Phantom reads occur when transaction 2 adds rows to a table between the multiple queries of transaction 1.

**The following isolation levels are supported by spring**

**ISOLATION\_DEFAULT**   
Use the isolation level of the underlying database.

**ISOLATION\_READ\_UNCOMMITTED**This is the lowest level of isolation and says that a transaction is allowed to read rows that have been added but not committed by another transaction. This level allows dirty reads, phantom reads and non repeatable reads.

**ISOLATION\_READ\_COMMITTED**   
This level allows multiple transactions on the same data but does not allow uncommited transaction of one transaction to be read by another. This level, therefore, prevents dirty reads but allows phantom reads and nonrepeatable reads. This is the default isolation setting for most database and is supported by most databases.

**ISOLATION\_REPEATABLE\_READ**   
This level ensures that the data set read during a transaction remains constant even if another transaction modifies and commits changes to the data. Therefore if transaction 1 reads 4 rows of data and transaction 2 modifies and commits the fourth row and then transaction 1 reads the four rows again then it does not see the modifications made by transaction 2. (It does not see the changes made in the fourth row by the second transaction). This level prevents dirty reads and non repeatable reads but allows phantom reads.

**ISOLATION\_SERIALIZABLE**  
This is the highest isolation level. It prevents dirty reads, non repeatable reads and phantom reads. This level prevents the situation when transaction 1 performs a query with a certain where clause and retrieves say four rows, transaction 2 inserts a row that forms part of the same where clause and then transaction 1 reruns the query with the same where clause but still sees only four rows (does not see the row added by the second transaction)

**Read Only**

The read only attribute specifies that the transaction is only going to read data from a database. The advantage is that the database may apply certain optimization to the transaction when it is declared to be read only. Since read only attribute comes in action as soon as the transaction starts, it may be applied to only those propagation settings that start a transaction. i.e. PROPAGATION\_REQUIRED,PROPAGATION\_REQUIRES\_NEW and PROPAGATION\_NESTED.

**Timeout**

Timeout specifies the maximum time allowed for a transaction to run. This may be required since transactions that run for a very long time may unnecessarily hold locks for a long time. When a transaction reaches the timeout period, it is rolled back. Timeout needs to be specified only on propagation settings that start a new transaction

**Rollback Rules**

It is also possible to specify that transactions roll back on certain exceptions and do not rollback on other exceptions by specifying the rollback rules.

**Spring Programmatic Transactions**

Programmatic transactions give a greater control to the user in setting up the transaction boundaries. It is possible to set up transaction boundaries to a part of the code (you will see later see that declarative transaction generally set transaction boundaries at method level). However, the transaction management code needs to be inserted into the program logic and hence is intrusive. Spring provides a template for programmatic transaction.   
The template class is org.springframework.transaction.support.TransactionTemplate. It uses default Transaction Attributes (PROPAGATION\_REQUIRED, ISOLATION\_DEFAULT, TIMEOUT\_DEFAULT, readOnly=false). It needs to be injected with a transaction manager, which should be an implementation of PlatFormTransactionManager (e.g. DataSourceTransactionManager). To change the transaction attributes use the following constructor.

public TransactionTemplate(PlatformTransactionManager transactionManager, TransactionDefinition transactionDefinition)

Use the DefaultTransactionDefinition and set the transaction attributes there.

The main method of the transaction template is the execute method with the following signature

|  |  |
| --- | --- |
|  | public <t> T execute(TransactionCallback<t> action) throws TransactionE  xception  </t></t> |

The TransactionCallback is a callback method that handles the actual transaction code. Please see the example below for more details. The transaction manager is configured as a bean in the application context and passed to the transactiontemplate.

**Spring Declarative Transactions**

TThe advantage of a declarative transaction is that all transaction demarcation happens through the xml and no extra coding needs to be done. It is also possible to declarative modify the transaction attributes without touching the code. Also changing the transaction managers is only a matter of modifying the transactionmanager bean. The disadvantage is that declarative transactions does not give a fine grained control over setting the transaction boundaries. However, in most cases, method level transaction settings suffice.

**Methodology**

the tx namesapce (http://www.springframework.org/schema/tx) defines xml elements for setting up transactions. The main element is <tx:advice>. Define an advice using this element. Create a pointcut (see the tutorial on Aspects to understand pointcuts) to define which beans participate in the transaction. Wire the advice and pointcut into an advisor. The advice element has an attribute called *transaction-manager*. This can be set to the Transaction Manager. However, if there is a bean whose id is transactionManager then it will be used by default and need not be set in the advice The advice element defines the transaction attributes of the methods using the <tx:method> element. The method element takes in a name which is the name of the method to which the transaction attributes are attached. It can use a wildcard character \* (e.g. add\* matches all methods starting with add).

**spring-config.xml**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="<http://www.springframework.org/schema/beans>"

xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns:aop="<http://www.springframework.org/schema/aop>"

xmlns:context="<http://www.springframework.org/schema/context>" xmlns:tx="<http://www.springframework.org/schema/tx>"

xsi:schemaLocation="

<http://www.springframework.org/schema/beans>

<http://www.springframework.org/schema/beans/spring-beans-3.0.xsd>

<http://www.springframework.org/schema/aop>

<http://www.springframework.org/schema/aop/spring-aop-3.0.xsd>

<http://www.springframework.org/schema/context>

<http://www.springframework.org/schema/context/spring-context-3.0.xsd>

<http://www.springframework.org/schema/tx>

<http://www.springframework.org/schema/tx/spring-tx-3.0.xsd>">

<aop:config>

<aop:advisor pointcut="execution(\* \*..AccountService.\*(..))"

advice-ref="transactionAdvice" />

</aop:config>

<tx:advice id="transactionAdvice" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="transferFunds\*" propagation="REQUIRED"

rollback-for="java.lang.Exception" />

</tx:attributes>

</tx:advice>

</beans>

**Spring Declarative Transactions Using Annotations**

The tx namespace also provides an element called <tx:annotation-driven> which can be used be define transactions using annotation. The Transaction Manager can be specified using the transaction-manager attribute, or if there is a bean names transactionManager then it will be used by default. The *annotation-driven* element specifies that all beans in the application context should be checked for the @Transactional annotation. The annotation can be set at class or method level. The transaction attributes can be specified along with the @Transactional annotation.

**spring-config.xml**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="<http://www.springframework.org/schema/beans>"

xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns:aop="<http://www.springframework.org/schema/aop>"

xmlns:context="<http://www.springframework.org/schema/context>" xmlns:tx="<http://www.springframework.org/schema/tx>"

xsi:schemaLocation="

<http://www.springframework.org/schema/beans>

<http://www.springframework.org/schema/beans/spring-beans-3.0.xsd>

<http://www.springframework.org/schema/aop>

<http://www.springframework.org/schema/aop/spring-aop-3.0.xsd>

<http://www.springframework.org/schema/context>

<http://www.springframework.org/schema/context/spring-context-3.0.xsd>

<http://www.springframework.org/schema/tx>

<http://www.springframework.org/schema/tx/spring-tx-3.0.xsd>">

<tx:annotation-driven transaction-manager="transactionManager" />

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

</beans>

**Service Class Method**

@Transactional(propagation=Propagation.REQUIRED, readOnly=false)

public void transferFunds(final Account fromAccount,

final Account toAccount, final Double transferAmount) {

fromAccount.debit(transferAmount);

toAccount.credit(transferAmount);

getAccountDao().update(fromAccount);

getAccountDao().update(toAccount);

}

transferFunds() method is used to successfully transfer funds from one account to another.   
Note that the *@Transactional* annotation is used to qualify the transferFunds() method .which tells Spring that this method should be executed within the scope of a transaction