### Spring Transaction

http://javawebtutorial.blogspot.com/2013/10/hibernate-transaction-management.html

**Transaction** represents a unit of work. Transaction is a set of SQL operations that needs to be either executed all completely or not at all.

Transaction is the process of performing multiple database operations as one unit with all nothing criteria.

I:e when all the database operations in the unit are successful then transaction is successful and should be committed. When any one database operation in the unit are failed then transaction are failed and should be rolled back.

|  |  |  |  |
| --- | --- | --- | --- |
| **ACID Properties** | **Transaction Concurrency Problem or Permitted Phenomenal** | **Isolation Level** | **Propagation** |
| Atomicity | Dirty Read | Transaction None | Propagation Required |
| Consistency | Non – Repeatable Read | Transaction Read Uncommitted | Propagation Required New |
| Isolation | Phantom Read | Transaction Read committed | Propagation Supports |
| Durability |  | Transaction Repeatable Read | Propagation Not Supported |
|  |  | Transaction Serializable | Propagation Mandatory |
|  |  |  | Propagation Never |
|  |  |  | Propagation Nested |

**When you implement transaction property in your application** **it guarantees ACID properties**.

A- Atomicity

C- Consistency

I- Isolation

D- Durability

**Atomicity** : It will give you a guarantee that the whole work will be performed as a unit. If one part of transaction fails, then entire transaction fails.

All success or none.

**Consistency** : Consistency is about ensuring that any transaction will bring the database from one valid state to another.

consistency represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.

Database constraints should not be violated.

**Isolation** : Many transactions may run concurrently. These concurrently running multiple transaction may disturb other transaction that is multiple transaction should run isolate.

one transaction should not affect another one.

Case A : Consider the case where multiple transaction running concurrently and using multiple rows of account table.

tx1 -> 101

tx2 -> 102

tx3 -> 103

1. withdraw from 101

2. withdraw from 102

3. withdraw from 103

Case B : Consider the case where multiple transaction running concurrently and using single account row of account table.

acc no: 99

1. transfer(withdraw)

2. bank teller(withdraw(deposite))

3. loan EMI(withdraw)

in case B you may get some problem, which is called as transnational concurrency problems.

Dirty read problem.

Repeatable read problem.

Phantom read problem.

To avoid these problems, you have to apply one of the **following required transactional isolation levels**.

READ\_UNCOMMITTED

READ\_COMMITTED

REPETABLE\_READ

SERIALIZABLE

**Durability** : Your enterprise data should be available for long time as long as your enterprise is running. You have to protect your data from crashes, failures, you have to implements your proper backup and recovery mechanism and proper logging mechanism.

it should be in database after commit.

**Dirty read problem** :

When transaction reads the dirty value(modified but not committed) **then you may get some inconsistent result**.

To avoid dirty read, you have to lock the column(cell).

To lock the column, you have to apply isolation level called READ-COMMITTED.

**Repeatable read problem** :

When a transaction is reading the same row repeatedly, **you may get different set of values in different reads**. this kind of problem is called repeatable read problem.

To avoid repeatable read problem, you have to lock the row.

To lock the row, you have to apply isolation level called REPEATABLE-READ.

**Phantom read problem** :

When a transaction is reading the set of rows repeatedly **you may get different set of rows in different reads** this kind of problem is called phantom read problem.

To avoid phantom read you have to lock the entire table.

To lock the table, you have to apply isolation level called SERIALIZABLE.

**Propagation -** Defines the propagation behaviour of the transaction. For example, you can specify a behaviour so that code can continue running in the existing transaction or the existing transaction can be suspended, and a new transaction created.

Default Propagation value is Propagation. REQUIRED.

propagation can have different possible value as below.

Propagation.REQUIRED – Support a current transaction, create a new one if none exists.

Propagation.REQUIRES\_NEW – Always create a new transaction and suspend the current transaction if already exist.

Propagation.MANDATORY – Support a current transaction, throw an exception if none exists.

Propagation.NESTED – Execute within a nested transaction if a current transaction exists.

Propagation.NEVER – Execute non-transactional, throw an exception if a transaction exists.

Propagation.NOT\_SUPPORTED – Execute non-transactional, suspend the current transaction if one exists.

Propagation.SUPPORTS – Support a current transaction, execute non-transactional if none exists.

Propagation.REQUIRED and Propagation.REQUIRES\_NEW is frequently used in real-time development.

https://www.javainuse.com/spring/boot-transaction-propagation

The **disadvantage of Propagation.REQUIRES\_NEW** is that even if the inner method fails to execute (because of some exception), the outer method commits the transaction. That causes inconsistency in data. If you use Propagation.REQUIRED, then if both inner/outer methods execute without fail, then only the data will be persisted to the database.

**Isolation** - This property defines the extent to which this transaction is isolated from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?

**isolation can have different possible value as below**.

Isolation.READ\_UNCOMMITTED – It allows dirty reads, non-repeatable reads, and phantom reads.

Isolation.READ\_COMMITTED – Dirty reads are prevented, allows non-repeatable and phantom reads.

Isolation.REPEATABLE\_READ – Dirty reads and non-repeatable prevented, phantom reads allowed.

Isolation.SERIALIZABLE – Dirty reads, non-repeatable reads, and phantom reads are prevented.

**Types of Transaction :**

**Local transaction :**

When a single database is participating in the transactional operation then it is called local transaction.

**Distributed transaction :**

When two or more database are participating in the transaction operations then it is called as distributed transaction.

**Types of Transaction** :

**Flat transactions** : A simple transaction that perform one or more operations as a unit operation.

**Nested Transactions** : as the word nested says one transaction under another transaction.

**Physical Transactions**: Are your actual JDBC transactions.

**Logical Transactions**: Are the (potentially nested) @Transactional-annotated (Spring) methods.

Methods are annotated with @Transactional so the Spring Hibernate transaction manager creates the required transactions and the respective sessions.

@Override

@Transactional

public void insertUser(User user) {

userDAO.insertUser(user);

}

**Programmatic transaction management**

– Here we need to write some extra code for transaction management. When we say some extra code what does it mean?

We need to take care of –

Creating Transaction reference

Begin transaction

Commit or rollback of the transaction.

public void insertUser(User user) {

Transaction transactionRef = entityManager.getTransaction()

try {

transactionRef.begin();

// business logic

userDAO.insertUser(user);

transactionRef .commit();

}

catch(Exception e) {

transactionRef.rollback();

e.printStackTrace();

}

}

**Declarative transaction management**

* No need to write extra code for getting a transaction, we can use annotations or XML-based approach to manage the transactions and we can avoid unnecessary code.
* @Transactional this is the key annotation which is responsible for the declarative transaction management. Apart from this annotation, we use @EnableTransactionManagement.
* @Transactional annotation used with class or method only.
* In Spring, @Transactional annotation is used for indicating a method run inside a database transaction. It can also be annotated on the class level which applies as a default to all methods of the declaring class and its subclasses
* Default settings are used for roll back of the transaction where the roll back happens if RunTime exception is thrown.

@Transactional(propagation=Propagation.REQUIRED,readOnly=false, isolation = Isolation.REPEATABLE\_READ)

public void addEmployee(Employee emp) {

empDAO.insertEmployee(emp);

}

* you now need to do two things:
  + Make sure that your Spring Configuration is annotated with the @EnableTransactionManagement annotation (In Spring Boot this will be done automatically for you).
  + Make sure you specify a transaction manager in your Spring Configuration (this you need to do anyway).

So, to get the @Transactional annotation working, all you need to do is this:

@Configuration

@EnableTransactionManagement

public class MySpringConfig {

@Bean

public PlatformTransactionManager txManager() {

return yourTxManager; // more on that later

}

}

@Bean

public SessionFactory sessionFactory(final HibernateEntityManagerFactory hemf) {

return hemf.getSessionFactory();

}

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/TEST" />

<property name="username" value="testuser" />

<property name="password" value="testpasswd" />

</bean>

<bean id="sessionFactory" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">

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

<property name="hibernateProperties">

<props>

<prop

key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>

<prop key="hibernate.show\_sql">true</prop>

</props>

</property>

<property name="packagesToScan" value="com.byteslounge.spring.tx.model" />

</bean>

<bean id="transactionManager"

class="org.springframework.orm.hibernate4.HibernateTransactionManager"

p:sessionFactory-ref="sessionFactory">

</bean>

import java.util.Properties;

import javax.sql.DataSource;

import org.hibernate.SessionFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;

import org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration;

import org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration;

import org.springframework.context.annotation.Bean;

import org.springframework.core.env.Environment;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

import org.springframework.orm.hibernate5.HibernateTransactionManager;

import org.springframework.orm.hibernate5.LocalSessionFactoryBean;

@SpringBootApplication

@EnableAutoConfiguration(exclude = { //

DataSourceAutoConfiguration.class, //

DataSourceTransactionManagerAutoConfiguration.class, //

HibernateJpaAutoConfiguration.class })

public class SpringBootHibernateApplication {

@Autowired

private Environment env;

public static void main(String[] args) {

SpringApplication.run(SpringBootHibernateApplication.class, args);

}

@Bean(name = "dataSource")

public DataSource getDataSource() {

DriverManagerDataSource dataSource = new DriverManagerDataSource();

// See: application.properties

dataSource.setDriverClassName(env.getProperty("spring.datasource.driver-class-name"));

dataSource.setUrl(env.getProperty("spring.datasource.url"));

dataSource.setUsername(env.getProperty("spring.datasource.username"));

dataSource.setPassword(env.getProperty("spring.datasource.password"));

System.out.println("## getDataSource: " + dataSource);

return dataSource;

}

@Autowired

@Bean(name = "sessionFactory")

public SessionFactory getSessionFactory(DataSource dataSource) throws Exception {

Properties properties = new Properties();

// See: application.properties

properties.put("hibernate.dialect", env.getProperty("spring.jpa.properties.hibernate.dialect"));

properties.put("hibernate.show\_sql", env.getProperty("spring.jpa.show-sql"));

properties.put("current\_session\_context\_class", //

env.getProperty("spring.jpa.properties.hibernate.current\_session\_context\_class"));

// Fix Postgres JPA Error:

// Method org.postgresql.jdbc.PgConnection.createClob() is not yet implemented.

// properties.put("hibernate.temp.use\_jdbc\_metadata\_defaults",false);

LocalSessionFactoryBean factoryBean = new LocalSessionFactoryBean();

// Package contain entity classes

factoryBean.setPackagesToScan(new String[] { "" });

factoryBean.setDataSource(dataSource);

factoryBean.setHibernateProperties(properties);

factoryBean.afterPropertiesSet();

//

SessionFactory sf = factoryBean.getObject();

System.out.println("## getSessionFactory: " + sf);

return sf;

}

@Autowired

@Bean(name = "transactionManager")

public HibernateTransactionManager getTransactionManager(SessionFactory sessionFactory) {

HibernateTransactionManager transactionManager = new HibernateTransactionManager(sessionFactory);

return transactionManager;

}

}

Now whenever you are using @Transactional on a bean, Spring uses a tiny trick. It does not just instantiate a UserService, but also a transactional proxy of that UserService.

It does that through a method called proxy-through-subclassing with the help of the Cglib library. There are also other ways to construct proxies (like Dynamic JDK proxies), but let’s leave it at that for the moment.

@Bean

public SessionFactory sessionFactory(final HibernateEntityManagerFactory hemf) {

return hemf.getSessionFactory();

}

If we have something wrong in our code data should not persist in the database.

**Read-only status** - A read-only transaction can be used when your code reads but does not modify data.

**readOnly** – Its value can be true or false.

@Transactional(readOnly = false)

**Timeout** - How long this transaction runs before timing out and being rolled back automatically by the underlying transaction infrastructure.

**@Transactional(timeout=60)**

Defaults to the default timeout of the underlying transaction system.

**@Transactional (rollbackFor=Exception.class)**

* Default is rollbackFor=RunTimeException.class
* In Spring, all API classes throw RuntimeException, which means if any method fails, the container will always rollback the ongoing transaction.
* The problem is only with checked exceptions. So, this option can be used to declaratively rollback a transaction if Checked Exception occurs.

https://www.javainuse.com/spring/boot-rollback

**@Transactional (noRollbackFor=IllegalStateException.class)**

Indicates that a rollback should not be issued if the target method raises this exception.

**@Transactional(isolation = Isolation.SERIALIZABLE)**

https://www.javainuse.com/spring/boot-transaction-isolation

// Using Transactional annotation we can define any isolation level supported by the underlying database.

@Transactional(isolation = Isolation.SERIALIZABLE)

@Transactional(isolation = Isolation.READ\_COMMITTED)

When using Transaction Isolation with Spring Boot, the default transaction isolation taken is that of the underlying database. So, for our spring boot application the default transaction isolation will be REPEATABLE\_READ since we are using MySQL database.

* READ\_COMMITTED is the default level with Postgres, SQL Server, and Oracle.
* REPEATABLE\_READ is the default level in Mysql. Oracle does not support REPEATABLE\_READ.
* Postgres does not support READ\_UNCOMMITTED isolation and falls back to READ\_COMMITED instead. Also, Oracle does not support and allow READ\_UNCOMMITTED

**@Transactional: Service or DAO Layer?**

The Service is the best place for putting @Transactional, **service layer should hold the detail-level use case behavior for a user interaction that would logically go in a transaction.**

Also, if you put @Transactional in DAO layer and if your **DAO layer is getting reused by different services** then it **will be difficult to put it on DAO layer as different services may have different requirements**.

Consider another example where your Service layer may call two different DAO methods to perform DB operations. If your first DAO operation failed, then the other two may be still passed and you will end up inconsistent DB state. Annotating a Service layer can save you from such situations.

**Transaction Logging**

A helpful method to understand transactional related issues is fine-tuning logging in the transactional packages. The relevant package in Spring is “org.springframework.transaction”, which should be configured with a logging level of TRACE.

The abstraction is via org.springframework.transaction.PlatformTransactionManager interface. Here is the snippet of the interface:

public interface PlatformTransactionManager {

TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

void commit(TransactionStatus status) throws TransactionException;

void rollback(TransactionStatus status) throws TransactionException;

}

**Some of the transaction managers are-**

DataSource Transaction manager - We can use DataSourceTransactionManager for simple JDBC persistence mechanism.

Hibernate Transaction manager – Hibernate transaction manager should be used when our application is using Hibernate.

Jdo Transaction manager – to use Java data object transaction manager

Jta Transaction manager – If our transaction is across multiple data sources than we need to use Java Transactions API transactions . Internally JTA implementation handles transaction responsibility.

org.springframework.orm.jpa.JpaTransactionManager — For JPA transactions

org.springframework.jdbc.datasource.DataSourceTransactionManager — For JDBC transactions

org.springframework.orm.hibernate5.HibernateTransactionManager — For Hibernate transactions and it binds with SessionFactory

org.springframework.transaction.jta.JtaTransactionManager — For JTA transactions

org.springframework.jms.connection.JmsTransactionManager — For JMS messaging transaction by binding JMS connection factory.