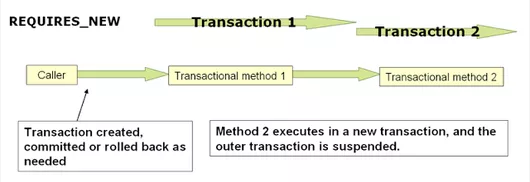
Global Transaction using JTA in Spring

**Global Transaction?**

Transaction Manager is an important aspect of any application. It is responsible for coordinating transaction related tasks across the application. Transactions can be related to single resource such as database operations or messaging system. But sometimes it may come over multiple resources also such as multiple database operations with different data sources or different resources as database and messaging system, which is called global transaction.

To manage transactions in any application, underlying framework on which application is build will provide manager to handle transaction related tasks. For example, Hibernate, Spring etc. If we are using single resource then easy to manage with given transaction manager such as HibernateTransactionManger or Spring’s TransactionManager. This we have already discussed in separate thread.

 If application required multiple resources which is independent of each other, then a separate manager is required to coordinate transactions among these. There are many api available to managing global transaction which is also termed as Distributed Transactions.

**Global Transactions in Spring?**

Spring given centralized transaction management PlatformTransactionManager which is further implemented to provide various transaction support. Mostly used implementations are:

* DataSourceTransactionManager
* JtaTransactionManager
* JpaTransactionManager

These are following component to provide basic idea of using JtaTransactionManager.

Click with ctrl to navigate to these topics.

1. Using JtaTransactionManager
2. Using JpaTransactionManager
3. Transaction Properties
4. **Using JtaTransactionManager**

In application, to use jta transaction manger, we need spring dependency for spring jdbc or orm with which transaction will be applied. Most importantly, we need global data source which will carry out db related operations. But global transactions are independent of local datasource, so we need container managed data source known as XA data source.

Spring Boot supports distributed JTA transactions across multiple XA resources using either an Atomikos or Bitronix embedded transaction manager. JTA transactions are also supported when deploying to a suitable Java EE Application Server. There are multiple others possible way to carry out distributed transactions.

To use Atomikos as transaction manager in spring, requires following dependency:

compile group: 'com.atomikos', name: 'transactions-jta', version: '4.0.3'

compile group: 'com.atomikos', name: 'transactions-jdbc', version: '4.0.3'

This will add atomikos transaction support for jta. Now, we will configure data sources which will be used by application in normal way.

**DataSource Configuration:**

AtomikosDataSourceBean : - The preferred class for using Atomikos connection pooling. Use an instance of this class if you want to use Atomikos JTA-enabled connection pooling. Using this bean, we can set name for this resource and xa data source also. For mysql connectivity, we need MysqlXADataSource datasource which will managed by Atomikos.

@Bean(initMethod = "init", destroyMethod = "close")

**public** DataSource xaLocalDataSource() {

MysqlXADataSource mysqlXaDataSource = **new** MysqlXADataSource();

mysqlXaDataSource.setUrl("jdbc:mysql://localhost:3306/test?useSSL=false&useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC");

//mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(true);

mysqlXaDataSource.setPassword("root");

mysqlXaDataSource.setUser("root");

AtomikosDataSourceBean xaDataSource = **new** AtomikosDataSourceBean();

xaDataSource.setXaDataSource(mysqlXaDataSource);

xaDataSource.setUniqueResourceName("localds");

**return** xaDataSource;

}

@Bean

@DependsOn("xaCustomDataSource")

**public** JdbcTemplate xaCustomJdbc() {

**return** **new** JdbcTemplate(xaCustomDataSource());

}

Above sample code will create a datasource which will be used in JdbcTemplate. Using that we can carry out any db tasks. That is special datasource which be taken care by atomikos jta.

Main part of configuration is transaction manager which will be implementation of PlatformTransactionManager. JtaTransactionManager needs two object as parameter which are:

**UserTransaction TransactionManager**

Its implementation is given by atomikos only, so need complex implementation. Below will show sample configuration of transaction manager.

@Configuration

@EnableTransactionManagement

**public** **class** XADBConfiguration{

@Bean

**public** PlatformTransactionManager txManager () **throws** Throwable {

**return** **new** JtaTransactionManager(userTransaction(), atomikosTxManager());

}

@Bean

**public** UserTransaction userTransaction() **throws** SystemException {

UserTransactionImp userTran = **new** UserTransactionImp();

userTran.setTransactionTimeout(10000);

**return** userTran;

}

@Bean

**public** TransactionManager atomikosTxManager() {

UserTransactionManager userTxMangager = **new** UserTransactionManager();

userTransactionMangager.setForceShutdown(**true**);

**return** userTransactionMangager;

}

}

This code will add jta transaction manager support to the application. Now, we can carry out task using earlier configured JdbcTemplate. Similarly, we can configure multiple datasources with different configurations.

For testing global transaction, write simple service method which may invoke several dao carring different jdbctemplate. As we can see, we added @EnableTransactionManagement to enable it and place @Transactional above service method with required transaction properties as propagation or isolation.

Now, make a rest call using below details, as per the dao code after inserting data into resources, it will cause a RuntimeException which inturn must be rolled back properly from those two resouces.

Make rest call, and verify the same in those data bases that records are inserted or not. To check negative output, comment @Transactional of service method and do the same. This time even after application throws RuntimeException, records are inserted in both the datasources.

Rest uri: **POST** <http://localhost:8096/customer/save>

Json value : {

"customerName": "tran2",

"emailid": "tran2@zeta.com",

"address": "delhi"

}

Read more: <https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-jta.html>

1. **Using JpaTransactionManager**

* To use JpaTransactionManager, we need JPA implementation. For this, we can go for Spring Data JPA or Hibernate also. Required dependencies for using spring data jpa is:

compile group: 'org.springframework', name: 'spring-orm, version:

'4.3.10.RELEASE'

compile("org.springframework.boot:spring-boot-starter-data-jpa")

* This will add all the implemented PlatfromTransactionManager to support various implementations. One of those is JpaTransactionManager.
* To user spring data jpa or hibernate-jpa implementation, we need to have EntityManager. As in core hibernate we use sessionfactory, JPA has EntityManager.
* To get EntityManager, we need define EntityManagerFactory which will be used to inject EntityManager. Below code will create the same:

@Bean

**public** LocalContainerEntityManagerFactoryBean factoryBean() {

LocalContainerEntityManagerFactoryBean factoryBean = **new** LocalContainerEntityManagerFactoryBean();

factoryBean.setDataSource(dataSource());

factoryBean.setPackagesToScan("com.rudra.aks.multitx.domain");

factoryBean.setJpaVendorAdapter(**new** HibernateJpaVendorAdapter());

factoryBean.setJpaProperties(jpaProperties());

**return** factoryBean;

}

@Bean

**public** DataSource dataSource() {

DataSourceBuilder builder = DataSourceBuilder.create();

builder.driverClassName("com.mysql.cj.jdbc.Driver");

builder.url("jdbc:mysql://localhost:3306/test");

builder.username("root");

builder.password("root");

**return** builder.build();

}

* Next step is to define the transaction manager which is JpaTransactionManager. This manager will take entitymanagerfactory as argument. So, we will pass previously created bean as argment. Below code is sample for the same:

@Bean

**public** PlatformTransactionManager transactionManager() {

JpaTransactionManager txManager = **new** JpaTransactionManager();

txManager.setEntityManagerFactory(factoryBean().getObject());

**return** txManager;

}

* Now we can make any db operation as previously discussed. If transaction management is enable using @EnableTransactionManager and annotating the service method with @Transactional then this operation will taken care by JpaTransactionManager.
* To test the functionality, make few rest calls with proper json data.

**POST** : <http://localhost:8096/customer/save>

{

"customerid": 1,

"customername": "grov1",

"emailid": "demo@zeta.com",

"address": "hyd"

}

Now we will see after getting exception, data is not persisted in database.

1. **Transaction Properties**

Along with above discussed transaction manager, we shall be using @Transactional annotation. This annotation is used to provide transaction behavior for current operation under the transaction. These are mainly used transaction behavior:

1. **Propagation** :- Controls the creation of transaction that will take care of given code for execution.
   1. **REQUIRED :-** Indicates that the target method can not run without an active tx. If a tx has already been started before the invocation of this method, then it will continue in the same tx or a new tx would begin soon as this method is called.
   2. **REQUIRES\_NEW :-** Indicates that a new tx must start every time the target method is called. If already a tx is going on, it will be suspended before starting a new one. Inner transaction may commit or rollback independently of the outer transaction, i.e. the outer transaction will not be affected by the inner transaction result.
   3. **SUPPORTS :-** Indicates that the target method can execute irrespective of a tx. If a tx is running, it will participate in the same tx. If executed without a tx it will still execute if no errors. Methods which fetch data are the best candidates for this option.
   4. **NEVER :-** Indicates that the target method will raise an exception if executed in a transactional process. This option is mostly not used in projects.
   5. **MANDATORY :-** Indicates that the target method requires an active tx to be running. If a tx is not going on, it will fail by throwing an exception. It requires an active tx otherwise tx fails.
2. **Isolation** :- This property controls effect of one transaction on another in concurrent transaction processing. If not specified the default isolation of database is used.
   1. **READ\_UNCOMMITTED :-** A transaction can read rows that have been added but not committed by another transaction. It caused dirty read.
   2. **READ\_COMMITTED** :- This will resolve dirty read problem. In this, a transaction will read only the committed data of another transaction also.
   3. **REPEATABLE\_READ :-** The above property will not phantom read problem but resolves non-repeatable reads which is if one transaction reads a set of records two time, it should be same both the times, even though another transaction modifies the same before second read.
   4. **SERILIAZABLE** :- This is the highest isolation level. It prevents dirty reads, non-repeatable reads and phantom reads. Phantom read is – when we query for records based on some conditions and for multiple read getting different number of records. Difference is caused by other transaction adding or removing data for same condition. This property will prevent everything.
3. **rollbackFor** :- Specifies the exceptions that will cause the rollback of transaction.