**2. Declarative management mechanism:**

|-Example Project Architecture

|-Order of project classes creation

|-spring bean configuration files in project

|-Tx Flow

|-Spring support to work with Tx's

|-Transactional Aspect Advices

|-Working with Default Exception in tx's

|-Difference between the Checked-Exception and Unchecked Exception through the Use Case

|-Throwing user Checked Exception in transactionality

|-Throwing user Un-Checked Exception in transactionality

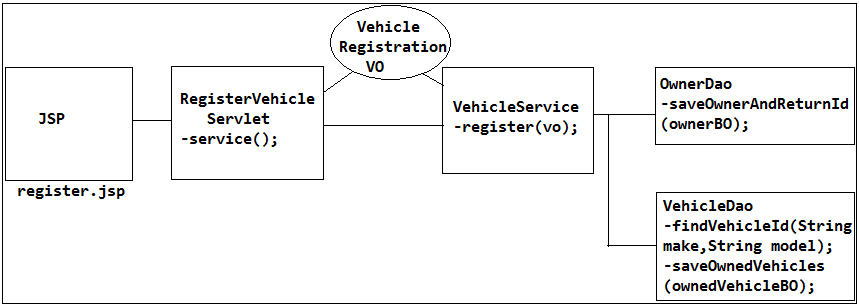
|-Difference between checked exception and UnChecked exception while working with Spring Transactions

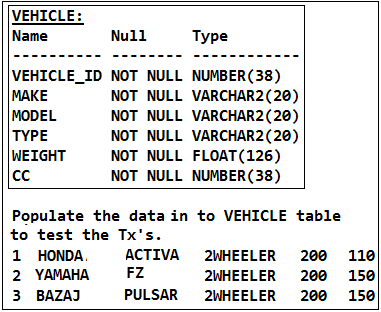
|-General problems while working Tx or Importing write Spring xsd

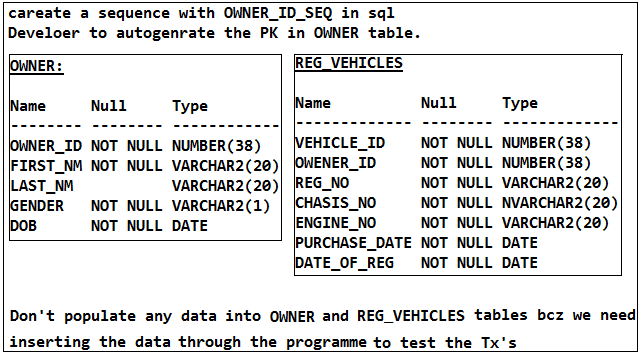
|-Choosing between programmatic and declarative transaction management

|-Wrapping Up/Conclusion

**Example Project Architecture:**







**Order of project classes creation:**

In order to create Dao we need BO, In order to create service we need VO hence we need to follow this order to create the project. That means BO, Dao, VO, service, controller then test class.

**Spring bean configuration files in project:**

🡪In a project multiple classes will be there like controllers, services, dao's so if declare all these classes as beans in an single spring bean config file then it becomes difficult manage and to modify in future that is the reason we will configure multiple spring bean config file based on the nature of the roles/functionality of the classes and all these config files will be imported/aggregated as part of single bean config file called as application-context.xml so that if can load application-beans.xml then automatically loads all the spring bean config files so that we can manage easily config file and we can identify and modify easily.

application-context.xml

|-persistence-beans.xml

|-service-beans.xml

|-controller-beans.xml

|-util-beans.xml

|-aop-beans.xml

|-performance-beans.xml

|-cache-beans.xml

|-log-beans.xml

persistence-beans.xml- Which contains all the persistency related components as bean like Dao's Datasources, jdbc templates etc.

🡪We are referring or importing all the beans in application-context.xml that is the reason it called as application config file.

🡪How many spring bean config files are there with in ur project and how did u managed them and how do pass them to create the IOC?

We create one root application context file where we will have the references to all the other config by importing them.

🡪Can we import multiple spring bean config file and if it is ok is there any order of import or not?

Yes, we can import spring bean config files in another spring bean config file but this import should be done in specific order bcz if services beans are referring the beans that are there in the persistence beans then 1st we need to declare persistence benas and import it 1st then we need to import the services beans in the application-context.xml that means dependent beans file be created 1st and target beans file created 1st then import 1st dependency beans then target beans file will be imported in the application-context.xml i.e order of instantiation should be matched with order of import.

🡪Did u create any obj in your project manually or not?

No we never create the obj directly rather we injected using spring Dependency Injection (DI)

🡪If one Dao throwing an Exception as EmptyResultDataAccessException then i don't wanted thow an Exception rather i wanted to return null how do we need to do this?

**Ans:** Put try and catch and return null

🡪How r u managing the transaction as part of u r application/project?

Explain the Project Architecture and then explain

**Tx Flow:**

All the connections should be committed at a time and all should be roll-back if any one of the operation is fail which is called Transactionality, for this we need to write the try-catch so that we can rollback the if any exception raised in any one of the operation so that in the catch or we can rollback the operation which is called as Tx which we need to write in the service layer bcz service performs the tx by calling multiple Dao's.

That means we have to write Tx related logic (try-catch with commit and rollback) in the service layer.

Instead of we maintaining the transactionality we can impose Spring tx which will takes imposing transactionality.

So we need to use Tx-manager/Tx-Co-ordinator with Resource Manager and Resource (dataSource).

If we wanted to implement Local Tx then we have to configure Local Tx related Tx manager that is provided by the spring.

If we wanted to implement Global tx then we need to configure Global Tx related Tx manager that is provided by the spring.

So TxManager is going to talk to the ResourceManager and he is going to talk to the Resources hence we need to pass dataSource as input to the TxManager so that he is going to open the connection and manages the Tx in combination with ResourceManager and inturn ResourceManager is going to talk to the Resource by taking the conn from the TxManager and intimates to the Txmanager whether sucess or failure so TxManager is the ultimate one to commit or rollback the operations.

Spring Local Tx is going to implement by the spring itself with same concept.

If we wanted impose Global Tx then Spring Tx manager is going to talk with JEE Tx's (JTA).

So if we configure Tx manager by passing dataSource as input then Tx-Manager is going to talk to the dataSource (DriverManagerDataSource) by setting autoCommit=false so dataSource will creates the conn with autoCommit as false and gives the conn to the TxManager now TxManager will mainatains that conn in the ThreadLocal level and gives conn to the ResourceManager to communicate with Resources.

In order to impose Tx we need to tell to the TxManager in which classes and in which methods we wanted to impose the tx.

So in the register() method of VehicleService which will talks to the multiple Dao's so we call ownerId = ownerDao.saveOwnerAndReturnId(ownerBO); in the service then ResourceManager will gets the conn from Txmanager and performs the operation by through the Resource which will executes jdbcTemplplate.excute() method (one resource represents one operation with that conn) so resource (commit will not happen by Resourcemanager bcz autoCommit=false set by the dataSource) and then we called vehicleId = vehicleDao.findVehicleId() and gets the vehicleId and sets to ownedVehicleBO obj and calls vehicleDao.saveOwnedVehicles (ownedVehicleBO); by getting same conn or one more conn from the dataSource and calls ResourceManager if it is another conn (dataSource may give same conn or may create one more conn in the ThreadLocal and gives to the Tx-manager) and if all are successfully executed then Tx-Manager will intimates to the Tx-Co-ordinator all are successfully executed asking him to commit finally Tx-Co-ordinator will commits (or rollback in case of any one is falied exception raised) all at one single shot.

Similarly if we have multiple service classes with multiple methods then we can configure those methods so that txManager will knows that where he wanted to apply the tx which is called as cross-cutting logic so we can use aop to configure these transactionality.

**Note:**

Multiple connections will not represents Global Tx, rather multiple Resources (multiple dataSources of 2-different Appl-servers) will represents the Global Tx.

That means if we have one Resource (dataSource) which is called as Local Tx.

If we have multiple dataSources then that will represents multiple Resources of 2-appl servers then it is called as GlobalTx.

**Spring support to work with Tx's:**

1. DataSourceTransactionManager (To work with Local Tx's)

2. JtaTransactionManager (To work with Global Tx's)

**Ex:**

<bean id="dataSource"

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

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />

<property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />

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

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

</bean>

<bean id="transactionManager"

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

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

</bean>

The above we configured DataSourceTransactionManager has one Resource (Resource=dataSource) that will contains multiple connections (pool of connections). That means one "dataSource" represents one Resource. Hence it is called as Local Tx.

If we observe org.springframework.jdbc.datasource.DataSourceTransactionManager is an dataSource that is provided by spring based on the Jdbc to work with tx's that is the reason they named package as org.springframework.jdbc.datasource here we need to observe the "jdbc" in the package name.

If we wanted to work with Global Tx's we need to configure one more class that is provided by the spring which will works with Global Tx's so for that class we need to pass 2-dataSource obj's as params.

To work with Global tx's we need to use org.springframework.transaction.jta.JtaTransactionManager that has been provided by the spring, if we observe the package name org.springframework.transaction.jta the package name itself is telling that "jta" it is Global Tx. The class JtaTransactionManager which inturns talks with JEE-Container to work with Global Tx's.

<bean id="jtaTransactionManager"

class="org.springframework.transaction.jta.JtaTransactionManager"

depends-on="atomikosTransactionManager,atomikosUserTransaction">

<property name="transactionManager" ref="atomikosTransactionManager" />

<property name="userTransaction" ref="atomikosUserTransaction" />

<property name="allowCustomIsolationLevels" value="true" />

</bean>

If we observe here we configures 2-dataSources (we can configure multiple dataSources as well) of 2-appl servers which are holed by the JtaTransactionManager hence it Global Tx.

**Transactional Aspect Advices:**

So if we configure TxManager it will activates one-active tx and talks to the Resourcemanager and applies the tx on the specified service classes methods.

If any exception raised by the Dao's the exception will be propagated service layer and now the service layer should not eat the exception rather it has to propagate the exception to the TxManager so that he will rollback the tx.

The transaction logic is not only specific to one service class rather tx need to apply for multiple service classes and we need to begin tx before the method of the service layer stating that one active tx is activated and it has to be committed after the service class method success fully executed and it has to rollback if any exception raised in the service class method. That means this type transactionality we need to apply not only in one service class rather we need to apply in various service layer classes methods which are participating or which are doing the tx operation that means it is not a primary logic (bcz without tx also our code will works but it becomes inconsistence that means it is secondary logic bcz we need to apply in all the aop) rather it is a cross cutting logic hence we need to use aop to apply the tx.

That means we need to write Tx Aspect classe representing crosscutting logic that we wanted to apply.

So we need to write 2-advice methods in the tx aspect class one is

(i) Around Advice

(ii) Throws Advice

We need to write the AroundAdvice for to begin tx before going to call the target class method [register() method of our service class] and we need to commit after the target class method [register() method of service class] has been executed successfully that means it is an AroundAdvice.

[Begin tx, commit, roolbacking tx is only we need to apply in one service rather we need to apply multiple service classes]

We need to one more advice method called as Throws Advice bcz whenever the exception raised we need to rollback the tx in hence if write the Throws advice the if any exception raised then control goes to the afterThrowing() method where we need to rollback the tx which we need to in all the target class methods [service classes methods] hence we need to apply aop Throws Advice as well for tx's.

So now internaly spring will writtens 2-advices in Tx Aspect class by implementing form MethodInterceptor and ThrowsAdvice.

🡪What are the Advices we are going to apply tx's on our target classes?

(i) Around Advice

(ii) Throws Advice

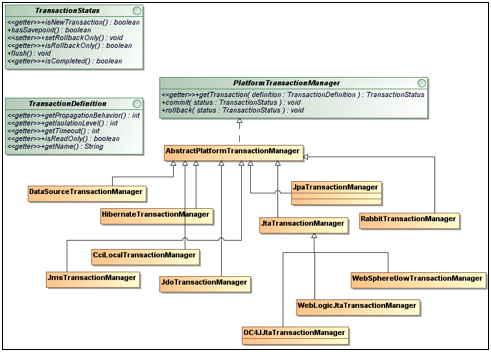
So we need to write the tx aspect containing AroundAdvice and Throws Advice but tx aspect we need to write not only in our project rather we need to write in multiple projects which is boiler-plate logic that's where spring itself has provided Tx Aspect which contains both advices.

That means our target class and aspect classes both must be a beans within the IOC. That means we need to know the spring Tx Aspect class and we need to configure as bean but is it difficult to remember and hence spring has provided one one tag called as <tx:advice> which will elaborates/expands this tag as class so internally spring is going to contains BeanFactoryPostProcessor which will reads meta-data before creating the IOC and creates Tx Aspect class as bean internally and places Tx Aspect also as bean with in the IOC internally so now our target class and Tx Aspect that has been provided by the spring also becomes bean in the IOC.

So we need to configure <tx:advice id="txAdvice" transaction-manager="transactionManager"> which is fixed names bcz we are not configuring the advice methods rather spring has configured the advice methods internally so if we write <tx:advice> internally 2-advice methods will be activated based on the pointciy that has been provide by spring so we need to refer that pointcut also same as spring specified pointcut reference hence we need to configure with **same id as <tx:advice id="txAdvice">** and we need to configure transaction-manager on this <tx:advice> bcz he is going to apply the tx-aspects on our target classes.

So we wanted work with Local Tx then we need to configure transaction-manager="transactionManager"> [here transactionManager is a bean that we configured as Local tx] and if we wanted to work with Global the Tx then we need to configure transaction-manager="jtaTransactionManager"> [here jtaTransactionManager is bean that we configured previously]

So we wanted to work with Jpa tx or Hibernate tx then we need to change the <transaction-manager=""> tag with corresponding tx manager and we no need to change any configuration and hence **spring is called as Single Uniform API which will works with any Frame Work tx's.**



Here we are configuring the transaction-manager="transactionManager" with in the <tx:advice> so that for this advice "transactionManager" will be injected into the Tx Aspect class of spring so that in that it will checks whether tx has been started or not bcz "transactionManager" tx manager is the ultimate one to rollback or commit the tx bcz transactionManager is not yet started the tx then aspect class should not apply the tx bcz we will get NullPointerExcepetion so it need to be a bean in the Tx Aspect so that on the "transactionManager" bean Spring Aspect will calls begin tx method on transactionManager and will applies the tx (in around advice and intimates to the Tx Manager thorugh the Resource Manager so finally Tx Manager is the ultimate one to commit or Rollbacks tx (in Throws Advice method) the operation on intimation by the ResourceManager.

We need to apply the only tx for DML operation only but not for the DQL (read-only operations) operations and hence we need to configure on which method we need to apply the tx.

interface PlatformTransactionManager {

public abstract void commit(TransactionStatus paramTransactionStatus);

public abstract void rollback(TransactionStatus paramTransactionStatus);

}

abstract class AbstractPlatformTransactionManager

implements PlatformTransactionManager {

}

class DataSourceTransactionManager extends AbstractPlatformTransactionManager implements

ResourceTransactionManager, InitializingBean {

private DataSource dataSource;

// setters & getters

}

This is called as Spring Local Tx.

aop-config.xml

<beans>

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

<tx:attributes>

<tx:method name="register" read-only="false"

rollback-for="VehicleNotFoundException" />

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut expression="within(com.dt.service.\*)" id="txPointcut" />

<aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />

<!-- Here advice-ref="txAdvice" and pointcut-ref="txPointcut" must be same name

that has been provided by spring which contains advice methods as advice

methods -->

</aop:config>

</beans>

So now if we call register() then it will directly not calls the register() method instead it applies the around advice method and follows the flow.

Test:

controller.register("DB", "Reddy", "M", new Date(), "HONDA", "STUNNER",

"AP09", "AC110", "CH09", new Date(), new Date());

/\*\*

\* Out put: (will throws Exception as VehicleNotFoundException (User

\* exception) So data will not gets persisted in OWNER and as well as

\* 'REG\_VEHICLES' bcz we applied the Spring tx the data will not

\* persisted in both tables bcz in DB 'VEHICLE' table MODEL STUNNER is

\* not there then also OWNER will gets but not committed and while

\* finding the vehicleId by calling findVehicleId() it throws exception

\* as EmptyResultDataAccessException stating that vehicleId not found

\* hence data will not gets inserted into REG\_VEHICLES table hence Tx

\* Manager will roll-backs whole tx which is consistence tx.

\*/

**Working with Default Exception in tx's:**

public class VehicleService {

public void register(VehicleRegistrationVO vehicleRegistrationVO) {

....

ownerId = ownerDao.saveOwnerAndReturnId(ownerBO);

vehicleId = vehicleDao.findVehicleId(

vehicleRegistrationVO.getMake(), vehicleRegistrationVO.getModel());

}

}

Here vehicleId = vehicleDao.findVehicleId() will throws **spring specific runtime** **exception** whenever vehicleId not found hence spring will understands and applies the tx without any problem bcz we are writing the try-catch and we are nor converting spring exception to our user exception so tx will be applied successfully.

<beans>

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

<tx:attributes>

<tx:method name="register" read-only="false" />

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut expression="within(com.dt.service.\*)" id="txPointcut" />

<aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />

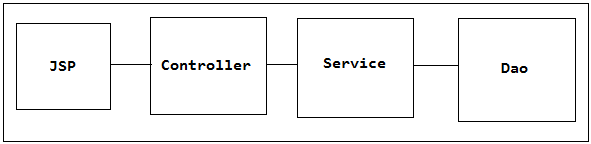
</aop:config>

</beans>

**Difference between the Checked-Exception and Unchecked Exception through the Use Case:**

We have a requirement where we need to process the payment order, where if card-no is not valid then also order can be re-tried. That we don’t wanted stop the transaction rather we wanted to preserve the transaction hence we need to throw Checked-Exception.

In Spring whatever the persistence tech it will throws Spring Jdbc Specific Unchecked Exception only (DataAccessException or its Subclass exception which are also UncheckedExceptions). So now service not required to catch it bcz DataAccessException and its child’s are Unchecked-Exception.



So if our service got any Unchecked-exception it should not commit rather it has to rollback bcz service not completed the tx successfully.

For example if we are booking a product through the e-commerce website then we will add all the product details to our cart that means then if we submit from the Jsp with Credit-CardNo, expiry and cvv the req comes to the Controller, now Controller need to check whether it is 16-digit or not then Service will calls one of the Dao by passing the productOrder info and payment info details then service must need to create the paymentOrder which contains productOrder and payments details and it has to call the PaymentDao to store the details in the DB for future reference even though the tx is not yet completed bcz if Tx unsuccessful and balance has been deducted then customer will complaint that balance has been deducted but tx was unsuccessful for this case we need to have a alternate data to check whether he has really attempted the payment or not, so after storing all the creditCard info and products info service will calls the Payment Gate-Way (Paypall) now the credit payment gate-way will throws InvalidCreditCardException stating that cardNo is not valid even though it is 16-digits bcz it not registered with any bank. So now our service will throws Checked InvalidCreditCardException then also Dao has to be still persist bcz we should not miss the order details but now we need to mark this as failed with same details so that we will have a way to respond to the end-user.

So if our service is throwing InvalidCreditCardException which is Checked-Exception then it will not rollbacks by default bcz this order can be re-tried that means we wanted preserve the state of the order or we need to lock the order and payment info even though it unsuccessful in such case we need to throw Checked-Exception.

But if the service is throwing the unchecked exception then it will by default will be marked as rollback. So if we don't wanted preserve the state of the tx then we need to throw the Unchecked-exception.

🡪When does the tx can be marked as roll-back can u plz tell me?

For all the exceptions our system should not rollback rather for unchecked exceptions only it has to rollback. As it is Unchecked-exception there is no point in committing the system hence we need to rollback bcz unchecked exceptions are non-recoverable.

For Checked-exception it will not rollbacks bcz checked-exception are recoverable so for checked-exceptions it will not rollbacks hence we have to tell whether it has be rollback or not bcz by default it will not roll-back for checked-exceptions.

🡪If our service is throwing an Checked Exception what will happen if we didn’t configure rollbackFor=””?

Ans: It will become In-Consistence, hence for Checked Exceptions we must configure for what checked exception Tx Manager has to rollback. But for Unchecked we no need to configure it by default rollbacks if exception raised even though we didn’t configure rollbackFor=”” for Unchecked Exception.

**Throwing user Checked Exception in transactionality:**

public class VehicleService {

ownerId = ownerDao.saveOwnerAndReturnId(ownerBO);

try {

vehicleId = vehicleDao.findVehicleId(

vehicleRegistrationVO.getMake(), vehicleRegistrationVO.getModel());

} catch (DataAccessException e) {

throw new VehicleNotFoundException("Vehicle make and model are not val", e);

}

}

If we wanted to throw our own CheckedException then we need to write exception in the catch block by catching with Spring DataAccessException then we can throw our own exception.

aop-beans.xml

<beans>

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

<tx:attributes>

<!-- Test: 1 -->

<!-- <tx:method name="register" read-only="false" /> -->

<!-- Observation: As our service class method register() is throwing **checked**

VehicleNotFoundException which is user defined exception hence for which

exception the tx should be rollback we have to specify otherwise tx will become

inconsistence. For example in the Test: 1 we didn't specified the exception but

we throwing our exception hence tx will become in-consistence bcz spring will

not know for which exception it has rollback hence it looks for **default spring**

**Unchecked - exception** but we didn't throw default exception of spring as it is

rather we thrown **our** **checked-exception** hence tx becomes inconsistence-- >

<!-- So if we throw **our checked exception** we have to specify for which

exception it has to roll-back so that tx will become consistence which has been

shown in Test: 2 -->

<!-- Test: 2 -->

<tx:method name="register" read-only="false"

rollback-for="VehicleNotFoundException" />

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut expression="within(com.dt.service.\*)" id="txPointcut" />

<aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />

</aop:config>

</beans>

For Test:2 the transaction will consistence bcz we specified for which exception transaction should be rollback bcz we are not throwing as it is default exception rather we are throwing user exception.

If we wanted to throw Our own Exception then we need to write exception in the catch block by catching with Spring DataAccessException then we can throw our own exception and we need to configure that exception in config for which exception the transaction should be rollbacked we need to configure.

**Throwing user Un-Checked Exception in transactionality:**

public class VehicleService {

ownerId = ownerDao.saveOwnerAndReturnId(ownerBO);

try {

vehicleId = vehicleDao.findVehicleId(

vehicleRegistrationVO.getMake(), vehicleRegistrationVO.getModel());

} catch (DataAccessException e) {

throw new VehicleNotFoundException("Vehicle make and model are not val", e);

}

}

If we wanted to throw our own **UnCheckedException** then we need to write exception in the catch block by catching with Spring DataAccessException then we can throw our own exception.

aop-beans.xml

<beans>

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

<tx:attributes>

<tx:method name="register" read-only="false" />

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut expression="within(com.dt.service.\*)" id="txPointcut" />

<aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />

</aop:config>

</beans>

If we wanted to throw **our own Unchecked Exception** then we need to write exception in the catch block by catching with Spring DataAccessException then we can throw our own exception. But we no need to configure for which exception spring has to rollback bcz All exceptions of Spring are UnCheckedExceptions hence if we write our own UnCheckedException then it will rollbacks automatically if any exception raises. This is happening bcz of our exception and spring exceptions both are Uncheched exceptions.

**Difference between checked exception and UnChecked exception while working with Spring Transactions:**

For **Checked Exceptions** the spring will not rollback if we didn’t configure rollbackFor="" within the config file.

For **Unchecked Exceptions** the spring by default rollbacks even though we didn’t configure the rollbackFor="" in the config file, this is happening bcz of spring specific exceptions and Our Exception both are Unchecked only.

Note that roll-backing is done by using the Tx Aspect Advice of the spring in the afterThrowing() method if it unchecked-exception by default, if it is checked-exception then it will not by default rollbacks hence we need to write for which it has to rollback.

**Note that if we want we can config for which exception we don’t wanted to rollback that facility is provided by the spring Tx.**

**General problems while working Tx or Importing write Spring xsd:**

While working the tx’s or anything we need to import write version related spring config xsd’s.

For example while we are working tx’s if we added spring 3.0 but while importing spring config xsd if we import wrong version of xsd which is related to spring 3.2 then we will exception as follows

Exception in thread "main"

**org.springframework.beans.factory.xml.XmlBeanDefinitionStoreException: Line 4 in XML document from class path resource [com/dt/common/application-context.xml] is invalid; nested exception is org.xml.sax.SAXParseException; lineNumber: 4; columnNumber: 132; cvc-elt.1: Cannot find the declaration of element 'beans'.**

**Caused by: org.xml.sax.SAXParseException; lineNumber: 4; columnNumber: 132; cvc-elt.1: Cannot find the declaration of element 'beans'.**

**Solution for the above problem:**

Import the write version of xsd which is compatible with spring config that means if we are working with spring 3.0 then we need to import the name spaces related to soring 3.0 only otherwise we will get above problem.

**Choosing between programmatic and declarative transaction management:**

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if we have a web application that require transactions only for certain update operations, we may not want to set up transactional proxies using Spring or any other technology. In this case, using the TransactionTemplate may be a good approach. Being able to set the transaction name explicitly is also something that can only be done using the programmatic approach to transaction management.

On the other hand, if our application has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure. When using the Spring Framework, rather than EJB CMT, the configuration cost of declarative transaction management is greatly reduced.

**Wrapping Up/Conclusion:**

Final conclusion is tx’s should not be committed or rollback individually rather service need to coomit or roll back with by treating it as single tx, but but if we use Spring tx we no need to commit or rollback rather spring will takes care of commit or rollback if we specify in which service class method we wanted apply the tx.

**WebSites:**

http://docs.oracle.com/cd/E13161\_01/tuxedo/docs10gr3/pgc/pgglob.html

http://docs.oracle.com/cd/E17802\_01/products/products/jta/jta-1\_0\_1B-doc/

https://www.progress.com/jdbc/resources/tutorials/understanding-jta/distributed-transactions-and-the-transaction-manager

http://integrationspot.blogspot.in/2011/03/jta-transactions-local-and-global.html

https://spring.io/blog/2011/08/15/configuring-spring-and-jta-without-full-java-ee/

http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/

https://spring.io/blog/2011/08/15/configuring-spring-and-jta-without-full-java-ee/

http://docs.spring.io/autorepo/docs/spring/4.2.x/spring-framework-reference/html/transaction.html#transaction-declarative

https://www.javacodegeeks.com/2013/07/spring-jta-multiple-resource-transactions-in-tomcat-with-atomikos-example.html

http://www.byteslounge.com/tutorials/spring-jta-multiple-resource-transactions-in-tomcat-with-atomikos-example