**Isolation Levels and Propagation Levels**

|-Isolation Levels

|-Introduction

|-Purpose of Isolation levels

|-Inconsistency problems

|-Use Case

|-Various/Different of Isolation Levels

|-Configuring Isolation levels using Config and Annotations

|-Isolation order

|-Config approach Vs Annotation Approach in Tx

|-Summary

|-Spring isolation with JPA

|-Propagation Levels

|-Introduction

|-Difference between Struts and Spring

|-Architecture of Spaning Tx across the service layers

|-Different Transaction Propagation Levels

|-Transactionl Attributes

|-Understanding with Combinations

|-Configuring Propagation levels Using Wild-Card patterns

|-Hibernate Tx Manager

**Isolation Levels:**

**Introduction:**

Transaction isolation level is a concept that is not exclusive to the Spring framework. It is applied to transactions in general and is directly related with the ACID transaction properties. Isolation level defines how the changes made to some data repository by one transaction affect other simultaneous concurrent transactions, and also how and when that changed data becomes available to other transactions. When we define a transaction using the Spring framework we are also able to configure in which isolation level that same transaction will be executed.

Now we need to know about the transaction isolation level provided by the Spring framework while working with Spring Tx's.

**Isolation Levels:**

The level at which accuracy and consistency on performing operation can be achieved based on isolation.

When an application is performing any persistence operation on DB (DML-operations), while performing this operations we may encounter with inconsistence issues like partial data has inserted or partial updated or partial data has been deleted that means parent has been deleted but child has been left as null all these are called as inconsistency issues while performing the DML operation using the applications.

In order to solve these type of inconsistency issues while performing the operation against the DB we need to use isolation levels. That means isolation levels will ensure whatever the operation that we are going to do those will ends up with accuracy or highest amount of consistency without any inconsistency.

**Purpose of Isolation levels:**

To avoid data inconsistency issues that we are going to face while performing the persistency operations we need to use Isolation levels.

That means we need to use one of the isolation level based on the problem we are going to face.

**Inconsistency problems:**

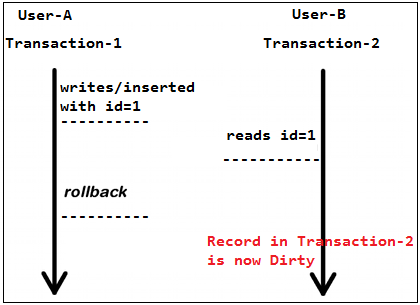
1. Dirty-Reads Problem

2. Non-Repeatable Reads Problem

3. Phantom Reads Problem

1. Dirty-Reads Problem (READ\_UNCOMMITTED):

Dirty-Read means a transaction may read data that is still uncommitted by other transactions. This constraint is very relaxed in what matters to transactional concurrency but it may lead to some issues like dirty reads. Let's see the following diagram:

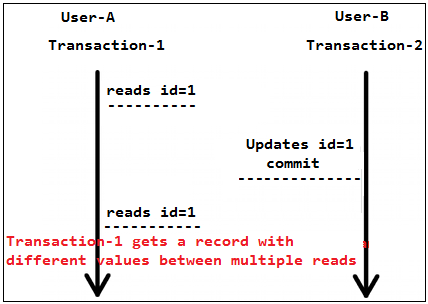


Within the organization or institute or college we have multiple employees are working to join the students by registering and with registered students some people are confirming the seats with that courses. Let us say we have one-active tx Transaction-1 (User-1) writes a record into the Student table with id=1. Meanwhile Transaction-2 (User-2) reads that same record id=1 before Transaction-1 commits. Later Transaction-1 decides to rollback and now we have changes in Transaction-2 that are inconsistent. This is a dirty read. Transaction-2 was running in READ\_UNCOMMITTED isolation level so it was able to read Transaction-1 changes before a commit occurred.

That means reading the un-committed changes of other transaction as part of another transactions which is dirty data called as Dirty-Read problem, which will causes the inconsistency issues.

2. Non-Repeatable Reads Problem / issue:

Let us say a transaction can't read data that is not yet committed by other transactions. This means that the dirty read is no longer an issue, but even this way other issues may occur. Let's see the following diagram:



In this example Transaction-1 reads some record. Then Transaction-2 writes data bu updating that same record and commits. Later Transaction-1 reads that same record again and may get different values because Transaction-2 made changes to that record and committed. This is a non-repeatable read.

Non-repeatable read means the record we are reading is not-same non-repeatable read issue/problem.

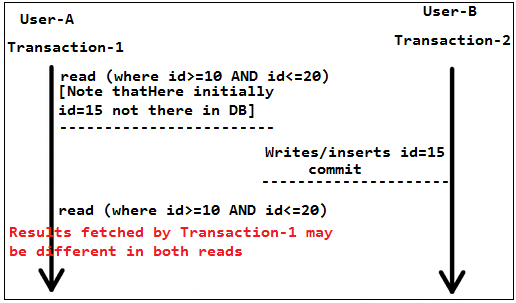
This is occurring bcz of we don't have any row level locking in an concurrent access/updates.

Use Case:

In Book-My show once we select the seat it should be lock otherwise seat-inconstancy will be occurred bcz same seat may book by other user.

3. Phantom Reads Problem:

If a transaction reads one record from the database multiple times the result of all those reading operations must always be the same. This eliminates both the dirty read and the non-repeatable read issues, but even this way other issues may occur. Let's see the following diagram:



In this example Transaction-1 reads a range of records. Meanwhile Transaction-2 inserts a new record in the same range that Transaction-1 initially fetched and commits. Later Transaction-1 reads the same range again and will also get the record that Transaction-2 just inserted. This is a phantom read: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).

Phantom means hunting new one or finding new one.

This is occurring bcz of we don't have any table level locking in an concurrent access/updates/inserts.

All these problems are occurring bcz of if we don't have any locking on that table or row. That means we need to apply the locking at various levels like record level locking or table level locking so that we can avoid these problems.

So avoid these data inconsistency problems we need to use Isolation Levels.

**Various/Different of Isolation Levels:**

So we can spring provided different isolation levels to avoid the inconsistency issues.

1. READ\_UNCOMMITTED

2. READ\_COMMITTED

3. REPEATABLE\_READ

4. SERIALIZABLE

5. DEFAULT

1. READ\_UNCOMMITTED:

If we use this isolation level then we all the 3-inconsistency problems will occurs. Bcz Tx Manager is going to start the tx in an READ\_UNCOMMITTED isolation mode. That means it will not solves any problems.

2. READ\_COMMITTED:

Used to avoid Dirty-Reads that means Tx manager will begins tx in READ\_COMMITTED isolation level. That means Tx Manager will asks the DB to read the data which is only committed data but uncommitted data.

It avoids Dirty-Reads problem but still we have Non-Repeatable Reads Problem and Phantom Reads Problem bcz we are not locking any record level locking or table level locking.

3. REPEATABLE\_READ:

It avoids Dirty-Reads problem and Non-Repeatable Reads Problem bcz it acquires the row level lock or record level lock but still we have Phantom Reads Problem.

4. SERIALIZABLE:

It is the high level isolation which will acquires table level lock.

SERIALIZABLE isolation level is the most restrictive of all isolation levels. Transactions are executed with locking at all levels (read, range and write locking) so they appear as if they were executed in a serialized way. This leads to a scenario where none of the issues mentioned above may occur, but in the other way we don't allow transaction concurrency and consequently introduce a performance penalty.

5. DEFAULT:

DEFAULT isolation level, as the name states, uses the default isolation level of the data store we are actually connecting from our application.

**Configuring Isolation levels using Config and Annotations:**

As Tx Manager is applying the tx's hence we need to tell to the Tx Manager which isolation has to be applied hence we need to configure this within the <tx:advice> bcz it has the <tx:method> tag.

In case of Annotation based tx using the @Transactional annotation we can define the isolation level of a Spring managed bean transactional method. This means that the transaction in which this method is executed will run with that isolation level so that we can avoid the inconsistency issues:

Configuration:

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

<tx:attributes>

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

</tx:attributes>

</tx:advice>

Here we specified read-only="false" that means if we want to modify some data in database its not possible with readOnly = true. But read-only="true" is ok if we want to retrieve some data set. So when it comes to UPDATE or INSERT use readOnly = false. Hence we need to specify readOnly = "false".

By default read-only=true.

(OR)

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

<tx:attributes>

<tx:method name="register" read-only="false" isolation="READ\_COMMITTED" />

</tx:attributes>

</tx:advice>

READ\_COMMITED is the second weakest of the four isolation levels defined by the SQL standard. Nevertheless (in spite of that), it is the default isolation level for many database engines, including SQL Server. This post in a series about isolation levels and the ACID properties of transactions looks at the logical and physical guarantees actually provided by read committed isolation.

We will apply READ\_COMMITED only in most of the applications, that means we didn't acquire any lock on record level locking or table level locking.

DEFAULT isolation level, as the name states, uses the default isolation level of the data store we are actually connecting from our application. READ\_COMMITED is default one which most of the DB's will follows. Hence DEFAULT=READ\_COMMITED. But Spring given as DEFAULT as isolation level for the sake of the developers understanding purpose.

If we wanted apply the READ\_COMMITED we don't need to do anything bcz it is the by default which most of the DB's will follows. That is the reason we will not see any isolation levels in the projects/applications.

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

<tx:attributes>

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

</tx:attributes>

</tx:advice>

🡪Where we need to write the isolation levels in case of config and annotations bases tx's and which type of isolation you guys applied?

We will apply READ\_COMMITED only in most of the applications.

If we wanted apply the READ\_COMMITED we don't need to do anything bcz it is the by default.

Annotaion Configuration Using @Transaction:

@Transactional(readOnly = false, isolation=Isolation.DEFAULT)

(OR)

@Transactional(readOnly = false, isolation=Isolation.READ\_COMMITTED)

(OR)

@Transactional(readOnly = false)

All these are serve same level of isolation only.

If we wanted to check read-only we need to use @Transactional annotation on top of the Dao to add declarative transactions.

@Transactional at class level wraps all method in transaction scope. The @Transactional annotation has several properties like readOnly, isolation, propagation, rollbackFor/rollbackForClassName, noRollbackFor etc that can be used to control how one transaction behaves and communicate with other transactions.

We use readOnly property of @Transactional to mark the method eligible only for making select queries when @Transactional(readOnly=true).

We use readOnly property of @Transactional to mark the method eligible for making update, insert queries in case @Transactional(readOnly=false).

By default readOnly is false, which means you can perform all the CRUD operations, change it to true will allow only select statements to be executed.

Other very useful properties of @Transactional is rollbackFor and noRollbackFor. By default Spring rolls back the transaction for exceptions of type RuntimeException or unchecked exceptions.

**Transaction Configuration using Wild-Card patterns:**

For example if we have register() methods like registerEmployee(), registerManager(), registerClerk(), registerCustomer() etc then we no need write <tx:method> multiple times rather we can configure using wild-card patterns easily if we follow the naming conventions for the methods on whom we wanted apply the tx's. So that we can avoid the writing bunch of config in the xml file.

If we want we can specify the for the methods who are going to perform read-only operations using <tx:method name=selectEmployee" read-only="true" /> but we no need to specify the read-only="true" bcz by default it is "true" only.

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

<tx:attributes>

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

<tx:method name="save\*" read-only="false" />

<tx:method name="select\*" read-only="true" /> [This we no need to specify bcz by default read-only="true"]

</tx:attributes>

</tx:advice>

But by default read-only="true" hence we no need to specify for all the methods read-only mode rather we need to specify only for tx methods only as read-only="false". use @Transactional(readoOnly = true) if you are performing a get/select and not making any changes, this means that no locks will be applied (which is more efficent).

For updates/inserts/deletions/saves/merges we need to use @Transactional (read-only="false") (when a lock is required):

For more Clarity see below Site

http://www.simplespringtutorial.com/springDeclarativeTransactions.html

**Isolation order:**

We need to write the methods isolation in an decreasing hierarchy order.

For example we have methods like

registerEmployee()

registerClerk()

registerManager() then

we wanted to apply isolation for registerClerk() as NON\_COMMITTED, for registerEmployee() and registerManager() we need to put NON\_REPEATABLEREAD then we should not configure as follows bcz for the methods isolation will applicable which is matched 1st time.

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

<tx:attributes>

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

isolation="NON\_REPEATABLEREAD" />

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

isolation="READ\_COMMITTED" />

</tx:attributes>

</tx:advice>

Here we configured 1st as for all the register\* methods apply the isolation as NON\_REPEATABLEREAD hence for all the methods isolation will applied as NON\_REPEATABLEREAD bcz this register\* wild card pattern includes all the methods. Hence we need to configure isaolation in an decreasing order as follows.

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

<tx:attributes>

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

isolation="READ\_COMMITTED" />

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

isolation="NON\_REPEATABLEREAD" />

</tx:attributes>

</tx:advice>

Here 1st looks for the registerClerk for this isolation as READ\_COMMITTED for next 2-methods isolation will be NON\_REPEATABLEREAD bcz it maches with register\* wild card pattern.

**Config approach Vs Annotation Approach in Tx:**

In order of isolation we need to specify in the decreasing order for config approach but this problem will not be there in the annotation config bcz we ill specify the isolation in each and every method.

Hence for tx we will use annotation approach only rather than config approach. For example if we have patterns like as follows then it is difficult to trace the isolation levels.

get\*

find\*

read\*

retrive\*

save\*

edit\*

create\*

modify\*

write\*

🡪Is there any way the order of <tx:attributes> that will effects isolation levels in the tx's?

Yes, in case of config approach we need to specify in the isolation decreasing order, but this problem will not be there in annotations.

**Summary:**

To summarize, the existing relationship between isolation level and read phenomena may be expressed in the following table:

Read Uncommitted: Allows dirty reads

Read Committed: Does not allow dirty reads

Repeatable Read: If a row is read twice in the same transaction, result will always be the same

Serializable: Performs all transactions in a sequence

Default: Use the default isolation level of the underlying data store.

|  |  |  |  |
| --- | --- | --- | --- |
| Isolation-level | Dirty-reads | Non-repeatable reads | Phantom reads |
| READ\_UNCOMMITTED | Yes | Yes | Yes |
| READ\_COMMITTED | No | Yes | Yes |
| REPEATABLE\_READ | No | No | Yes |
| SERIALIZABLE | No | No | No |

**Spring isolation with JPA:**

If you are using Spring with JPA we may come across the following exception when we use an isolation level that is different the default:

InvalidIsolationLevelException: Standard JPA does not support custom isolation levels - use a special JpaDialect for your JPA implementation at org.springframework.orm.jpa.DefaultJpaDialect.beginTransaction(DefaultJpaDialect.java:67)

To solve this problem you must implement a custom JPA dialect which is explained in detail in the following website:

http://www.byteslounge.com/tutorials/spring-transaction-isolation-tutorial

http://www.byteslounge.com/tutorials/spring-change-transaction-isolation-level-example

**Conclusion:**

Finally conclusion is we no need to perform tx’s in all the methods rather we need to apply tx's in specific classes of specific methods who are going to perform DML operation hence <tx:advice> need to ask the transaction-manager start tx in an read-only="false" modeand with READ\_COMMITTED isolation level so that tx manager starts the tx according to that.

**Propagation Levels**

|-Introduction

|-Difference between Struts and Spring

|-Architecture of Spaning Tx across the service layers

|-Different Transaction Propagation Levels

|-Transactionl Attributes

|-Understanding with Combinations

|-Configuring Propagation levels Using Wild-Card patterns

We need to discuss about the transaction propagation behaviours provided by the Spring framework.

Boundaries of the tx's will be decided by the Propagation level, that means defines how transactions relate to each other.

**Introduction:**

While dealing with Spring managed transactions the developer is able to specify how the transactions should behave in terms of propagation. In other words the developer has the ability to decide how the business methods should be encapsulated in both logical or physical transactions. Methods from distinct Spring beans may be executed in the same transaction scope or actually being spanned across multiple nested transactions. This may lead to details like how does the inner transaction outcome result affects the outer transactions. We will see the different propagation behaviours provided by Spring now.

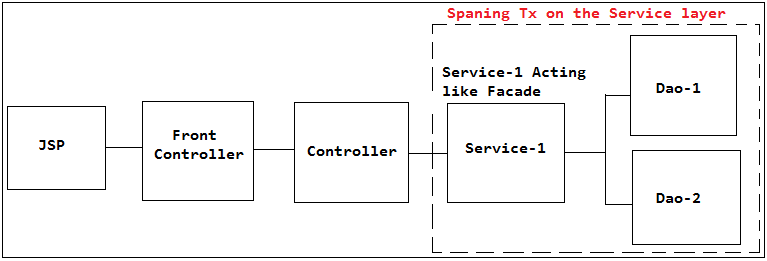
**Difference between Struts and Spring:**

Struts is a Web-Mvc frame work it will not provide any other tier features like managing the persistency tier, business tier, transactionality, dependency injection of the application apart from presentation tier aspects.

Spring is a frame work using which we can develop all the layers of our application.

**Architecture of Spaning Tx across the service layers:**

**Diagram: 1**



The jsp will sends the req to the controller, the controller will calls the service and it will do business operation but not the persistence operation hence it will talks to multiple Dao's(Here we completely eliminated the Business delegate that is tx's will takes care by spring tx-module, exception translations we no need to write, custom translations also we no need to write, mapping the data is also we no need to write (VO to BO), validation also handled gracefully hence no need to use Delegate that is the reason direclty we can write service).

Never a Dao will not calls multiple Dao's bcz if any Dao is calling another Dao it is difficult to migrate from one DB to another DB.

We have use case where we have multiple service classes which involves in the same transaction based on our business requirement then Controller should not talk to the multiple service layers bcz the amount of coupling between the business and presentation tier will becomes more.

If controller is talking to the multiple service layers then if any one of the service layer has modified then presentation tier has to be modified due to which presentation tier is tightly coupled with business tier.

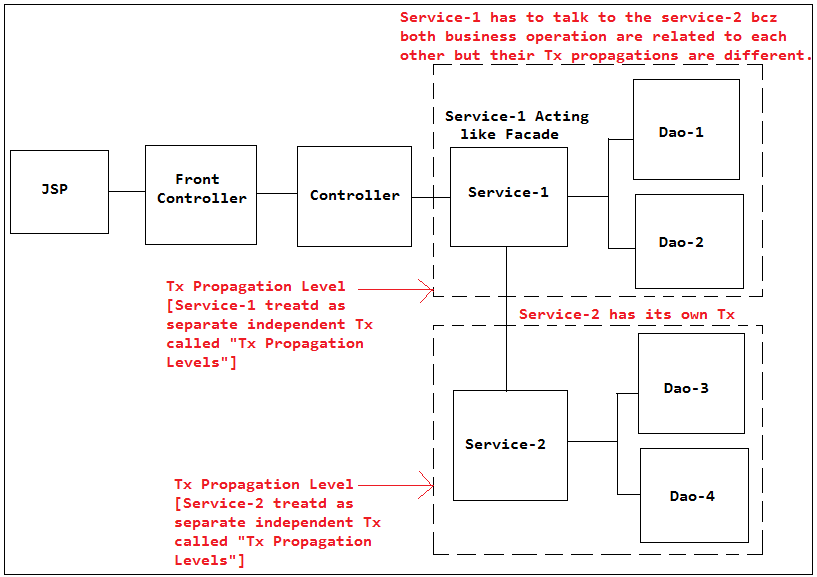
If controller is talking to the multiple service layer then it has to know the which classes has to be call 1st and which classes has to be call next to fulfil the business operation due to which presentation tier is going to be tightly coupled with business tier.

Hence the one-controller should not talk to the multiple service layers so that controller will be abstracted from the business layer how many classes are involved in the business operation. Hence the Controller should not (should never) talk to the multiple service classes rather it has to call one service layer method only so that migration efforts will become easy.

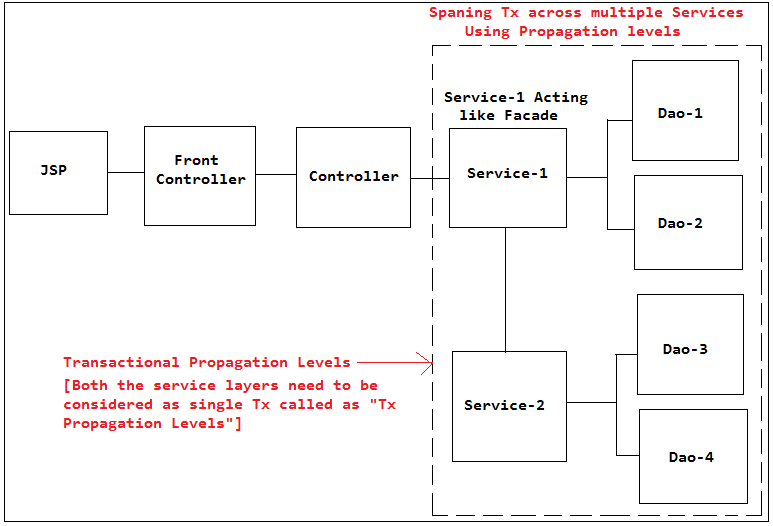
As controller is not talking to the multiple services but we have requirement where we need to span the tx across the service layers hence one service layer need to call another service layer in fulfilling the tx, hence the service which is calling another service will be acts like "Facade". So that service is layer that shields/abstracts the complexity in performing the business operations on the presentation tier components of our application this is main advantage of the service.

Hence one-Controller is responsible to handle only type one req only.

**Diagram: 2**



**Diagram: 3**



🡪How do u manage the transactions across the Dao's?

Ans: Using Service Layer.

🡪How do u manage the transactions across the services?

Ans: Using Transactional Propagation Levels.

🡪How do u manage the transactions across the Controllers?

Ans: We will never manage the Tx's with in the Controllers bcz presentation tier and business will become tightly coupled to each other. Hence one-Controller is responsible to handle only type one req only and we will not span tx in the controllers.

Hence we need to decide whether 2-service layers will fall under the same Tx or in an different Tx based on our business requirement and we need to apply the Tx Propagation Levels according to that, so deciding the transactional boundaries across the multiple service layers is possible by using "Transactional Propagation Levels".

Here in the diagram also Service-1 will calls Dao-1 and Dao-2 after this it has to call service-2 which will talks to the Dao-3 and Dao-4 that means once Service-1 has completed by calling Dao-1 and Dao-2 it has to call the service-2 with same Tx or it has to initiate new transaction will be decided by the Transactional Propagation Levels.

The Tx is beginning by the Tx Manager and <tx:advice> is telling to the Tx Manager on which class methods need to apply the tx hence it is the responsibility of the <tx:advice> to tell to the Tx Manager in which Propagation Level the tx has to begin on the transactional methods so that Tx Manager acts accordingly. Hence we need to configure Tx Propagation Levels also within the <tx:advice>.

**Different Transaction Propagation Levels:**

**Transactionl Attributes:**

In transaction interceptor that means on the tx method, we have to define which transaction’s attributes 'propagation behavior'should be use. It means if a transactional 'register()' method is called another service 'method', how the transaction should be propagated? Should it continue to run within the existing transaction? or start a new transaction for its own. This is called as Tx Propagation.

There are 7 types of transactional propagation attributes supported by Spring in the enum Propagation:

1. REQUIRED – Support a current/same transaction; creates one new active tx if not exists. That means joins tx.

2. REQUIRES\_NEW – Create a new transaction, suspending the current transaction if one exists. That means it always requires a new Tx for each service.

3. SUPPORTS – Support a current transaction; execute non-transactionally if none exists.

4. NOT\_SUPPORTED – Do not support a current transaction; rather always execute non-transactionally.

5. MANDATORY – Support a current transaction; throw an exception if no current transaction exists.

6. NEVER – Do not support a current transaction; throw an exception if a current transaction exists.

7. NESTED – Execute within a nested transaction if a current transaction exists, behave like REQUIRED else.

Flow:

So when the controller is calling the register() method on the service-1 layer then method call will not enters into the register() instead of the control goes to the Tx Aspect tx:advice it checks what is the read-only is false or ture then Tx isolation and Tx Propagation level and begin the Tx according to that.

**Understanding with Combinations:**

The combination given below are explained based on the **Diagram: 3**

Case: 1

Service-1 method: REQUIRED

Service-2 method: REQUIRED

In this aces if service-1 has active tx then service-2 also need to join within the same tx so if any exception raised within the service-2 Dao classes then all will be rollbacked.

Case: 2

Service-1 method: REQUIRED

Service-2 method: REQUIRES\_NEW

In this case if service-1 has active tx then service-2 will suspends (suspends means it will continue with same tx in the service-2 that service-1 tx will applicable service-1 only) the current tx and begins one more new active tx to perform the operation. So if any exception raised within the service-2 Dao classes then Service-2 only will be rollbacked but Service-1 will gets committed.

Use Case: 3

Loan Approval is normal actual Tx and Audit will be the another Tx where we need to use this Tx Propagation level. That means if Audit throws (Service-2) exception fails also the actual business operation (Service-1) should be committed.

Case: 4

Service-1 method: REQUIRED

Service-2 method: SUPPORTED

In this case if service-1 has active tx then service-2 will neither joins within the current tx nor begins new tx rather executes non-transactionally without any tx by suspending the current tx of the service-1.

Case: 4

Service-1 method: REQUIRED

Service-2 method: NEVER

In this case if service-1 has active tx then service-2 will neither joins within the current tx nor begins new tx and will not executes rather it will throws exception stating that call me without any tx.

Case: 5

Service-1 method: NEVER

Service-2 method: MANDATORY

This is mutually exclusive.

Similarly understand all the combinations.

In most of the cases, we may just need to use the REQUIRED. If we didn't configure any propagation level by default is REQUIRED.

If any exception raised by the Dao's we will get DataAccessException which is unchecked exception of spring.

Configuration Approach:

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

<tx:attributes>

<tx:method name="regi\*" read-only="true"

rollback-for="VehicleNotFoundException" propagation="REQUIRED" />

</tx:attributes>

</tx:advice>

Annotation Approach:

@Transactional(readOnly = false,

rollbackForClassName = { "VehicleNotFoundException" },

propagation = Propagation.REQUIRED)

(OR)

@Transactional(readOnly = false,

rollbackFor = VehicleNotFoundException.class,

propagation = Propagation.REQUIRED)

**Configuring Propagation levels Using Wild-Card patterns:**

In addition, we have to define the method to support this transaction attributes as well. The method name is supported wild card format, a register\* or save\* will match all method name start with regiater(.....) or save(.....).

🡪Can u plz explain what are Transactional Propagation Levels and how many are there with the help of architectural Diagram?

🡪What is the propagation level we need to use for this use case the interviwer will ask us?

Ans: Propagation.REQUIRED and which by default as well if don't configure.

🡪What is the Isolation level and under which isolation level ypu guys are working in tx?

🡪What is the Propagation level and under which Propagation level ypu guys are working in tx?

**Hibernate Transaction Manager:**

In Hibernate transaction, you need to use HibernateTransactionManager. If you only deal with pure JDBC, use DataSourceTransactionManager; while JTA, use JtaTransactionManager.