Spring Transaction Management:

Case 1: no transactional annotation == runtime exception with no active transaction present for synchronization

Case 2: Transactional annotation == creates a new transaction and inserts

Case 3: Transaction annotation with access modifiers (@Transaction annotation)

Private – Runtime Exception

Void – Runtime Exception

Protected- Runtime Exception

Public – creates a new transaction and inserts

Case 4: Nested Methods in same class

@Transaction public + (private,public,default,protected) – creates a new transaction and inserts

@Transaction private + (private,public,default,protected) – runtimeException

@Transaction default + (private,public,default,protected) – runtimeException

@Transaction protected + (private,public,default,protected) – runtimeException

Case 5: Nested Methods in same class with nested child methods also having transactional annotation

@Transaction public +@transactional (private,public,default,protected) – creates a new transactionboundary of child and inserts

@Transaction private + @transactional (private,public,default,protected) – runtimeException

@Transaction default +@transactional (private,public,default,protected) – runtimeException

@Transaction protected + @transactional (private,public,default,protected) – runtimeException

Case 5: list iterating from service layer calling transactional boundary in dao layer with exception in entry in list

Ans : records are written until the exception occurs . if it is handled then other valid records inserts else termination

Case 6: list iterating from service layer calling dao layer(transaction layer in service layer) with exception in entry in list

Writes data into temp cache deletes when there is exception no records will persist

Case 7: list iterating from service layer calling dao layer(transaction layer in service layer) with exception in entry in list with catch block

Writes data into temp cache deletes when there is exception no records will persist

Case 8: transactional boundary with checked exceptions and sub classes for list of data with exception records .

Valid records persist . invalid records will not be persited use (@Transactional(rollback= exception.class)

Different transactional levels:

* **REQUIRED**: NONE->T1, T1->T1
* **REQUIRES\_NEW**: NONE->T1, T1->T2
* **MANDATORY**: NONE->Exception, T1->T1
* **NOT\_SUPPORTED**: NONE->NONE, T1->NONE
* **SUPPORTS**: NONE->NONE, T1->T1
* **NEVER**: NONE->NONE, T1->Exception

Case 9: Handling exceptions for invalid records and write other valid records

@Transactional(noRollbackFor= NullPointerException.class)

Case 10: T1->T1-> T2

If t2 creates a new transaction boundary whatever commits happen in that boundary will not be rollback by parent transaction

Case 11: what if we create service object with new instead of autowired.

Transactional proxy will not be created if we create with new because spring ioc is not creating the object

Isolation Levels:

* **Dirty Read –** A Dirty read is the situation when a transaction reads a data that has not yet been commited.For example, Let’s say transaction 1 updates a row and leaves it uncommited, meanwhile Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
* **Non Repeatable read –** Non Repeatable read occurs when a transaction reads same row twice, and get a different value each time. For example, suppose transaction T1 reads a data. Due to concurrency, another transaction T2 updates the same data and commit, Now if transaction T1 rereads the same data, it will retrieve a different value.
* **Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different. For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that matches the search criteria for transaction T1. If transaction T1 reexecutes the statement that reads the rows, it gets a different set of rows this time.

**Isolation levels**

1. **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet commited changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.
2. **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction hold a read or write lock on the current row, and thus prevent other rows from reading, updating or deleting it.
3. **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and write locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non repeatable read.
4. **Serializable –** This is the Highest isolation level. A serializable execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently ececuting transactions appears to be serially executing.

**Default =(read committed)**