Spring transaction management:

The transaction can be defined with ACID properties.

* **Atomicity:**It means either all the operations (insert, update, delete) inside a transaction take place or none. Or you can say, all the statements (insert, update, delete) inside a transaction are either completed or rolled back.
* **Consistency:** It means, whatever happens in the middle of the transaction, this property will never leave your database in a half-completed state.
  + If the transaction completed successfully, then it will apply all the changes to the database.
  + If there is an error in a transaction, then all the changes that already made will be rolled back automatically. It means the database will restore to its state that it had before the transaction started.
  + If there is a system failure in the middle of the transaction, then also, all the changes made already will automatically rollback.
* **Isolation:** Every transaction is individual, and One transaction can’t access the result of other transactions until the transaction completed. Or, you can’t perform the same operation using multiple transactions at the same time. We will explain this SQL acid property in a separate article.
* **Durability:** Once the transaction completed, then the changes it has made to the database will be permanent. Even if there is a system failure, or any abnormal changes also, this SQL acid property will safeguard the committed data.

**Declarative transaction management:**

If we use annotation-based approach we can use @Transactional and if we use the XML-based approach we need to configure DataSourceTransactionManager or any other transaction manager in XML as a bean.

@Transactional this is the key annotation which is responsible for the declarative transaction management. The annotation itself having a lot of things so let’s see what the possible attribute are we can define inside @Transactional annotation. By using this attributes, we can customize our transaction.

Let’s see possible attributes of @Transactional annotation.

@Transactional(isolation = Isolation.DEFAULT,

               propagation=Propagation.REQUIRES\_NEW,

               readOnly=true,

               noRollbackFor =ArithmeticException.class,

               timeout = 30000,

               value="txManager2",

               rollbackFor = { Exception.class },

               rollbackForClassName = {"Exception"},

Another point here isolation and propagation attribute has different values. Now it seems a little bit tricky. Yes until unless we are not sure about attributes names and possible values (and what it does exactly, it is not easy to implement in real time development). The good news we use only isolation and propagation frequently in real-time development (Although this is not hard and fast rule depends on your requirement).

Let’s see all attribute/property one by one.

**propagation –** 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-transitionally, throw an exception if a transaction exists.
* Propagation.NOT\_SUPPORTED – Execute non-transitionally, suspend the current transaction if one exists.
* Propagation.SUPPORTS – Support a current transaction, execute non-transitionally if none exists.

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

Note: Default Propagation value is Propagation. REQUIRED

**isolation –**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.

Default isolation value is Isolation.DEFAULT.

A **dirty read** occurs when one transaction is permitted to **read** data that is being modified by another transaction.

A **non-repeatable** read is one in which data read twice inside the same transaction. It cannot be guaranteed to contain the same values between the two reads. Depending on the isolation level, another transaction could updated the values between the two reads.

**Phantom read** happens when one transaction executes a query twice and it gets a different number of rows in the result set each time. This happens when a second transaction inserts a new row that matches the WHERE clause of the query executed by the first transaction.

**readOnly –** Its value can be true or false. Go through this [**post**](https://netsurfingzone.com/spring/transactional-readonly-true-example-in-spring-boot) for more details.

@Transactional(readOnly = false)

@Transactional

If we don’t provide any attribute @Transactional, then the default value will be @Transactional(readOnly = false).

Generally, we use @Transactional(readOnly = true) for search or retrieval operation to make sure we can only perform the read-only operation.

Ex:

@Transactional(readOnly = true)

public List<Student> getAllStudents() {

List<Student> studentResponse = (List<Student>) studentRepository.findAll();

return studentResponse;

}

If you use the method level to create or update We have the same response as earlier in DB but Does any record persist in DB? No there will not any record in DB. Since we are using @Transactional(readOnly = true), means we can perform the read-only operation, we can’t save any record in DB

Note: @Transactional(readOnly = false), we can perform read/write operation.

@Transactional(propagation=Propagation.REQUIRED) – If there is not an existing transaction it will create a new transaction. In case if there is already an existing transaction it will not create a new transaction.

@Transactional(propagation=Propagation.REQUIRES\_NEW) – Even if there is already an existing transaction it will create a new transaction i.e it will always create a new transaction.