**Spring Transactions - 2024**

**What is transaction management**

A database transaction is a sequence of actions that are treated as a single unit of work.

**Global transactions (also called a distributed transaction)**

A global transaction will span multiple resources (this might be two different database, database and message queue, etc). In a global transaction you will write to the DB and send a message over a queue. Global Transaction is an application server managed transaction.

**Local transactions**

Local transactions are resource-specific, such as a transaction associated with a JDBC connection. Local Transaction is resource specific transaction (for example [Oracle Transactions](http://docs.oracle.com/cd/E11882_01/server.112/e10713/transact.htm#CNCPT016)) and application server has nothing to do with them.

The primary difference is that **local transaction are single JVM level** while **global transaction are at multiple JVM level** and hence global transactions need different design approach

**ACID Property**

**Atomicity**: Atomicity means that an entire transaction either takes place all at once or it doesn’t occur at all. It means that there’s no midway.

**Consistency:** Consistency means Correctness, data should not be lost under any circumstances and there cannot be partial update or logical corruption. **Also consistency in data is called referential integrity**. Referential integrity means relationships between tables remain consistent. Consistency means database must be consistent before and after transaction.

**Isolation:** Each transaction is independent of other transactions. Any reads or writes performed on the database will not be impacted by other reads and writes of separate transactions occurring on the same database. Isolation is a guarantee that concurrently running transactions should not interfere with each other.

**Durability:** All transaction results are permanently preserved. Ensures that the results of the transaction are permanently stored in the system. The modifications must persist even in case of power loss or system failures.

**Database anomalies**

**Dirty read:** read the uncommitted change of a concurrent transaction

**Dirty reads [Read uncommitted] occur when**:

* Transaction A inserts/updates a row into a table.
* Transaction B reads the that row.
* Transaction A rolls back.

Transaction B may have done work to the system based on the row inserted by transaction A, but that row never became a permanent part of the database.

**T1**: **insert into user (id, age) values (1, 21);** or **update user set age = 21 where id = 1;**

**T2: select age from user where id = 1;**

**T1: Rollback**

In the above case T2 gets the value as 21.

**Solution 🡺 @Transactional(isolation = Isolation.READ\_COMMITTED)**

**READ\_COMMITTED is the default level with Postgres, SQL Server, and Oracle**

**Nonrepeatable read**: get different value on re-read of a row if a concurrent transaction updates the same row and commits.

**Non-Repeatable reads [Read committed] occur when**:

* Transaction A reads a row.
* Transaction B changes the row.
* Transaction A reads the same row a second time and gets the new results.

**T1: select \* from user where id = 1;**

**T2: update user set age = 51 where id = 1; commit**

**T1:** select \* from user where id = 1;

**Solution 🡺 @Transactional(isolation = Isolation.REPEATABLE\_READ)**

**REPEATABLE\_READ eliminates both the dirty read and the non-repeatable read issues.**

**REPEATABLE\_READ is the default level in Mysql. Oracle does not support REPEATABLE\_READ**

**Phantom read:** get different rows after re-execution of a range query if another transaction adds or removes some rows in the range and commits.

**Phantom reads occur when**:

* Transaction A reads all rows that satisfy range query
* Transaction B inserts an additional row that satisfies the WHERE clause.
* Transaction A re-evaluates the WHERE condition and picks up the additional row.

**T1: select \* from user where age between 10 and 30;**

**T2: insert into user (id, age) values (3, 27);**

**T1: select \* from user where age between 10 and 30;**

**Solution 🡺 @Transactional(isolation = Isolation.SERIALIZABLE)**

**SERIALIZABLE** isolation level is the most restrictive of all isolation levels. Transactions are executed with locking at all levels. This solves Dirty Reads, NonRepeatable Read and Phantom Read.

**@Transactional(**

**propagation = Propagation.*REQUIRED*,**

**isolation = Isolation.*SERIALIZABLE*,**

**noRollbackFor = NullPointerException.class,**

**noRollbackForClassName = {"Abcd", "PQRS"},**

**rollbackFor = IndexOutOfBoundsException.class,**

**rollbackForClassName = "Abcd",**

**readOnly = true,**

**timeout = 3,**

**value = "SomeValue")**

**Spring transaction propagation**

When a method with a transaction calls other methods, transactions are propagated from one method to another. It is the distribution/dissemination of transactional behaviour among the various methods.. Defines how transactions relate to each other. specify how transactions should propagate across method calls and nested transactions.

**PROPAGATION\_REQUIRED – Support a current transaction; create a new one if none exists.**

**PROPAGATION\_SUPPORTS – Support a current transaction; execute non-transactionally if none exists.**

**PROPAGATION\_MANDATORY – Support a current transaction; throw an exception if no current transaction exists.**

**PROPAGATION\_REQUIRES\_NEW – Create a new transaction, suspending the current transaction if one exists.**

**PROPAGATION\_NOT\_SUPPORTED – Do not support a current transaction; rather always execute non-transactionally.**

**PROPAGATION\_NEVER – Do not support a current transaction; throw an exception if a current transaction exists.**

**PROPAGATION\_NESTED – Execute within a nested transaction if a current transaction exists, behave like PROPAGATION\_REQUIRED else.**

* **Required (default)**: My method needs a transaction, either open one for me or use an existing one → getConnection(). setAutocommit(false). commit().
* **Supports**: I don’t really care if a transaction is open or not, i can work either way → nothing to do with JDBC
* **Mandatory**: I’m not going to open up a transaction myself, but I’m going to cry if no one else opened one up → nothing to do with JDBC
* **Require\_new:** I want my completely own transaction → getConnection(). setAutocommit(false). commit().
* **Not\_Supported:** I really don’t like transactions, I will even try and suspend a current, running transaction → nothing to do with JDBC
* **Never:** I’m going to cry if someone else started up a transaction → nothing to do with JDBC
* **Nested:** It sounds so complicated, but we are just talking savepoints! → connection.setSavepoint()

**Required Vs Requires\_New**

Using REQUIRES\_NEW is only relevant when the method is invoked from a transactional context; when the method is invoked from a non-transactional context, it will behave exactly as REQUIRED - it will create a new transaction. A practical example is given below.

**Suppose you're in charge of implementing a signup service in which a confirmation e-mail is sent to the user. You come up with two service objects, one for enrolling the user and one for sending e-mails, which the latter is called inside the first one. For example something like this:**

**@Service 🡸 /\* E-Mail Service \*/**

**@Transactional(Propagation=REQUIRES\_NEW)**

**class EmailService{**

**...**

**void sendMail(User user){**

**try{**

**... // Trying to send the e-mail**

**}catch( Exception)**

**}**

**}**

**@Service 🡸 /\* Sign Up service \*/**

**@Transactional(Propagation=REQUIRED)**

**class SignUpService{**

**...**

**void SignUp(User user){**

**...**

**emailService.sendMail(User);**

**}**

**}**

You may have noticed that the second service is of propagation type REQUIRES\_NEW and moreover, chances are it throws an exception (SMTP server down , invalid e-mail, or other reasons). You probably don't want the whole process to roll back, like removing the user information from a database or other things; therefore you call the second service in a separate transaction.

**In case of Mandatory (PROPAGATION\_MANDATORY): You define your DAO classes this way:**

**/\* User DAO \*/**

**@Transactional(Propagation=MANDATORY)**

**class UserDAO{**

**// some CRUD methods**

**}**

Meaning that whenever a DAO object, and hence a potential access to DB, is created, we need to reassure that the call was made from inside one of our services, implying that a live transaction should exist; otherwise, an exception occurs. Therefore the propagation is of type MANDATORY.

**Enum Isolation**

**DEFAULT:** Use the default isolation level of the underlying datastore.

**READ\_COMMITTED:** prevents dirty; non-repeatable reads and phantom reads can occur.

**REPEATABLE\_READ:** Prevents dirty reads and non-repeatable reads; phantom reads can occur.

**SERIALIZABLE:** Prevents dirty reads, non-repeatable reads and phantom reads.

**READ\_UNCOMMITTED:** dirty reads, non-repeatable reads and phantom reads can occur.

**How @Transactional works internally**

So when you annotate a method with @Transactional, Spring dynamically creates a proxy that implements the same interface(s) as the class you're annotating. And when clients make calls into your object, the calls are intercepted and the behaviors injected via the proxy mechanism.

**@Transactional** **only rolls back transactions for unchecked exceptions**. For checked exceptions and their subclasses, it commits data. So although an exception is raised here, because it's a checked exception, Spring ignores it and commits the data to the database, making the system inconsistent.

**Miscellaneous Use Cases**

**Case -1 : Use of RuntimeException and explicit handling of exception**

@Transactional

public void createPersonAddress(Person person, Address adrs) {

createPerson(person);

try {

createAddress(adrs);

} catch (Exception e) {

e.printStackTrace();

}

}

In both the tables, data will be saved. It does not matter whether you have written **Propagation.*REQUIRES\_NEW*** or **Propagation.Required** or simply **@Transactional. The default is Propagation.Required.** The reason behind is we are handling the exception so data will be saved.

**Case -2 : No handling of exception**

@Transactional

public void createPersonAddress(Person person, Address adrs) {

createPerson(person);

createAddress(adrs);

}

@Transactional

public void createAddress(Address adrs) {

personAdrsDAO.createAddress(adrs);

throw new RuntimeException("Unwanted exception ...");

}

In this case data will not be saved as createAddress throws a RumtimeException. So entire transaction is rolled back.

**Case -3 : Use of Propagation.*REQUIRES\_NEW***

@Transactional

public void createPersonAddress(Person person, Address adrs) {

createPerson(person);

createAddress(adrs);

}

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void createAddress(Address adrs) {

personAdrsDAO.createAddress(adrs);

throw new RuntimeException("Unwanted exception ...");

}

In this case data will not be saved , no new transaction will be started as it is inside the public method, same class and same proxy.

**Case -4: Handling of exception with Propagation.REQUIRES\_NEW**

@Transactional

public void createPersonAddress(Person person, Address adrs) {

createPerson(person);

createAddress(adrs);

}

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void createAddress(Address adrs) {

try {

personAdrsDAO.createAddress(adrs);

throw new RuntimeException("Unwanted exception ...");

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

Data will be saved in both tables, as we are handling exception. Propagation setting has no impact.

**Case -5 : Use a private method and create a separate class to handle**

@Transactional

public void createPersonAddress(Person person, Address adrs) {

createPerson(person);

try {

updateAddress(adrs);

} catch (Exception e) {

e.printStackTrace();

}

}

@Autowired

ApplicationContext context;

private void updateAddress(Address adrs) {

AnotherServiceImpl another = (AnotherServiceImpl) context.getBean("anotherService");

another.createAddress(adrs);

}

@Service

public class AnotherServiceImpl {

@Autowired

private JdbcTemplate jdbcTemplate;

@Transactional(propagation = Propagation.REQUIRES\_NEW)

public void createAddress(Address adrs) {

String insertQuery = "insert into address (city) values(?)";

Object[] params = new Object[] {adrs.getCity() };

jdbcTemplate.update(insertQuery, params);

System.out.println("Address saved successfully");

throw new RuntimeException("Unwanted exception ...");

}

}

In this case, person table will be populated with the data and address will not be populated.

If we change to only @Transactional in the AnotherServiceImpl class, then both the tables will not be populated with data. This is a special case, refer to Flipkart transaction use case above.

**Spring Transaction Comes into Picture when you have multiple services (@Service classes) and methods there are annotated with @Transactional**

**Use Case-1**

@Service

public class AddressServiceImpl {

@Autowired

private AddressRepository adrsRepo;

@Transactional

public void saveAddress(Address adrs) {

adrsRepo.save(adrs);

}

}

@Service

public class PersistenceServiceImpl implements PersistenceService {

@Autowired

private EmpRepository empRepo;

@Autowired

private AddressServiceImpl adrsService;

@Transactional

@Override

public void saveBothEmpAddress(Employee emp, Address adrs) {

empRepo.save(emp);

adrsService.saveAddress(adrs);

throw new NullPointerException("Deliberate Exception ...");

}

}

In the above case, data will not be persisted in both the tables. Because, both the service class methods are annotated with only @Transactional.

**Use Case - 2**

In the below case, method is annotated with @Transactional(propagation = Propagation.REQUIRES\_NEW).

@Service

public class AddressServiceImpl {

@Autowired

private AddressRepository adrsRepo;

**@Transactional(propagation = Propagation.REQUIRES\_NEW)**

public void saveAddress(Address adrs) {

adrsRepo.save(adrs);

}

}

If we call the method saveBothEmpAddress() of PersistenceServiceImpl class, data will not be saved in employee table but **data will be saved in address table**. The reason is AddressServiceImpl is a separate service class and its method has been annotated with **@Transactional(propagation = Propagation.REQUIRES\_NEW).** Here it is important that Propagation works for multiple services.

**Use Case-3**

@Transactional

@Override

public void saveBothEmpAddress(Employee emp, Address adrs) {

empRepo.save(emp);

saveAddress(adrs);

throw new NullPointerException("Deliberate Exception ...");

}

@Transactional(propagation = Propagation.REQUIRES\_NEW)

public void saveAddress(Address adrs) {

adrsService.saveAddress(adrs);

}

and AddressServiceImpl looks like this.

@Service

public class AddressServiceImpl {

@Autowired

private AddressRepository adrsRepo;

public void saveAddress(Address adrs) {

adrsRepo.save(adrs);

}

}

In this case no data will be saved.

**Use Case - 4**

@Transactional

@Override

public void saveBothEmpAddress(Employee emp, Address adrs) {

empRepo.save(emp);

saveAddress(adrs);

throw new NullPointerException("Deliberate Exception ...");

}

@Transactional(propagation = Propagation.REQUIRES\_NEW)

public void saveAddress(Address adrs) {

adrsRepo.save(adrs);

}

In this case, it is the same service class, methods are different and propagation are different. So data will not be saved.