Prepared by Radha V krishna

TRANSACTIONS IN SPRING BOOT

WHAT IS @TRANSACTIONAL

- We can use *@Transactional* to wrap a method in a database transaction.
- It allows us to set propagation, isolation, timeout, read-only, and rollback conditions for our transaction

WHAT IS TRANSACTION PROPAGATION?

- Transaction Propagation indicates if any component or service will or will not participate in transaction and how will it **behave if the calling calling component/service already has or does not have a transaction created already.**

PROPAGATION LEVELS

- **REQUIRED**
- **REQUIRES_NEW**
- **SUPPORTS**
- **MANDATORY**
- **NOT_SUPPORTED**
- **NEVER**
- **NESTED**

REQUIRED

REQUIRED is the default propagation. Spring checks if there is an active transaction, and if nothing exists, it creates a new one. Otherwise, the business logic appends to the currently active transaction:

REQUIRES_NEW

When the propagation is *REQUIRES_NEW*, Spring suspends the current transaction if it exists, and then creates a new one:

SUPPORTS

For *SUPPORTS*, Spring first checks if an active transaction exists. If a transaction exists, then the existing transaction will be used. If there isn't a transaction, it is executed non-transactional:

MANDATORY

When the propagation is *MANDATORY*, if there is an active transaction, then it will be used. If there isn't an active transaction, then Spring throws an exception:

NOT_SUPPORTED

If a current transaction exists, first Spring suspends it, and then the business logic is executed without a transaction:

NEVER

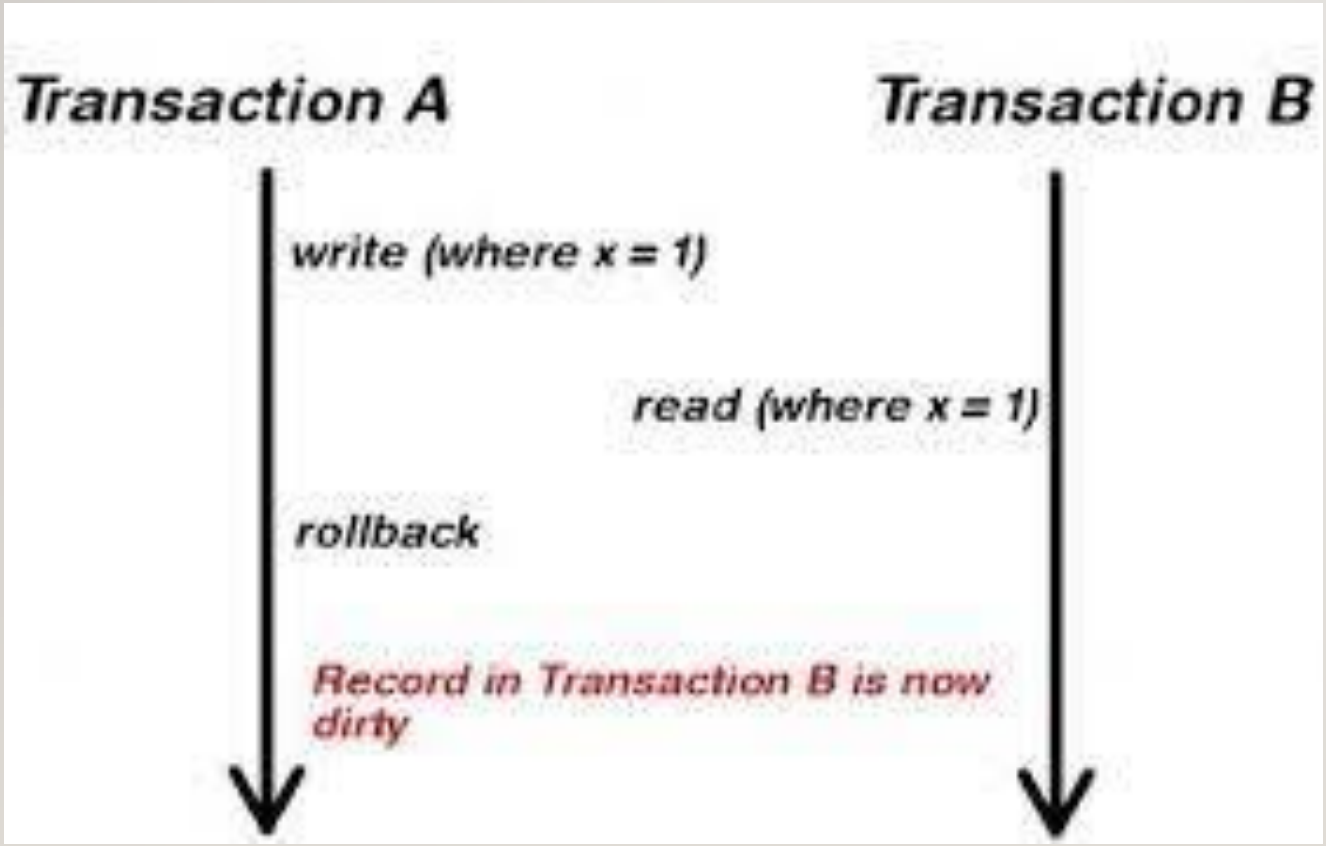
For transactional logic with *NEVER* propagation, Spring throws an exception if there's an active transaction:

NESTED

For *NESTED* propagation, Spring checks if a transaction exists, and if so, it marks a save point. This means that if our business logic execution throws an exception, then the transaction rollbacks to this save point. If there's no active transaction, it works like *REQUIRED*.

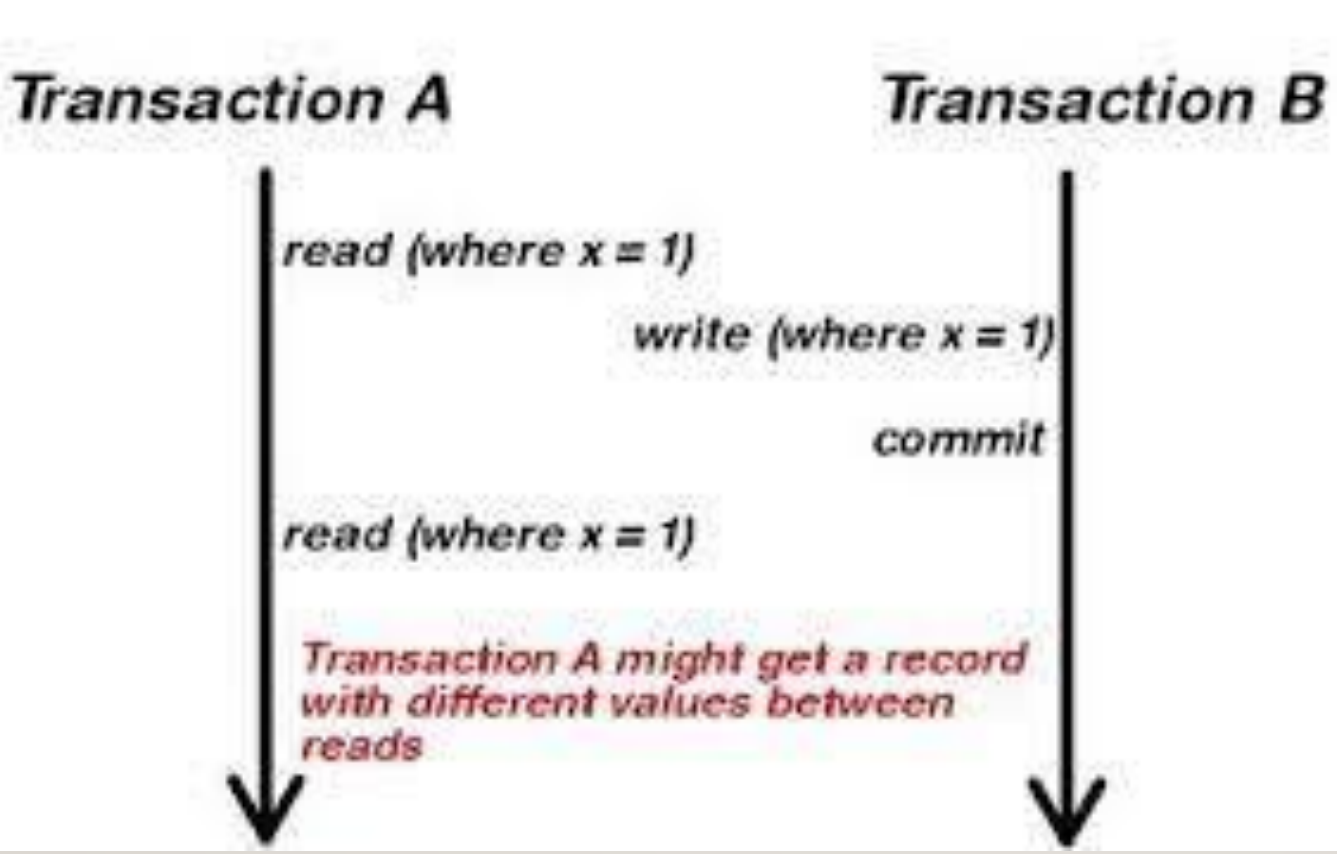
Isolation is one of the common ACID properties: Atomicity, Consistency, Isolation, and Durability. Isolation describes how changes applied by concurrent transactions are visible to each other.
Each isolation level prevents zero or more concurrency side effects on a transaction:

Dirty read: read the uncommitted change of a concurrent transaction



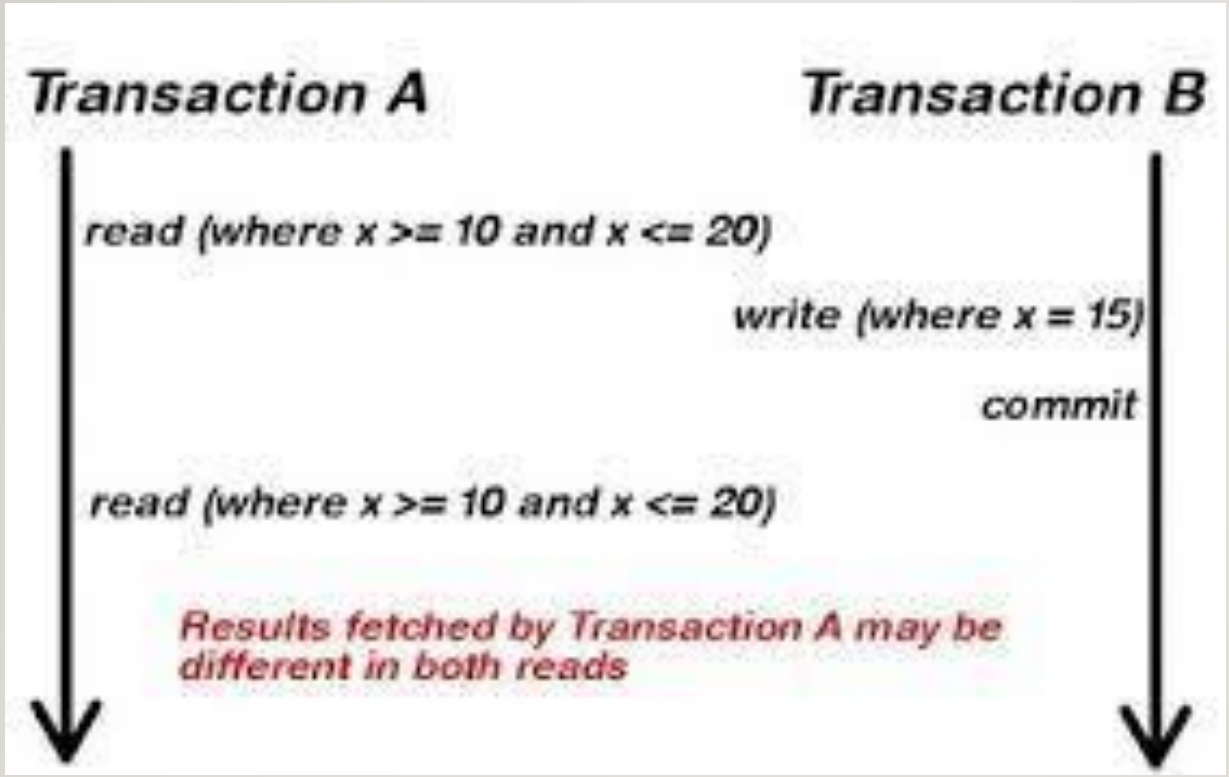
Non Repeatable Reads

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



Phantom Reads

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



Setting Isolation levels

```
@Transactional(isolation = Isolation.READ_UNCOMMITTED)
```

READ_UNCOMMITTED is the lowest isolation level and allows for the most concurrent access.
As a result, it suffers from all three mentioned concurrency side effects. A transaction with this isolation reads uncommitted data of other concurrent transactions. Also, both non-repeatable and phantom reads can happen. Thus we can get a different result on re-read of a row or re-execution of a range query.

```
@Transactional(isolation = Isolation.READ_COMMITTED)
```

The second level of isolation, *READ_COMMITTED,* prevents dirty reads.
The rest of the concurrency side effects could still happen. So uncommitted changes in concurrent transactions have no impact on us, but if a transaction commits its changes, our result could change by re-querying.

```
@Transactional(isolation = Isolation.REPEATABLE_READ)
```

The third level of isolation, *REPEATABLE_READ,* prevents dirty, and non-repeatable reads. So we are not affected by uncommitted changes in concurrent transactions.
Also, when we re-query for a row, we don't get a different result. However, in the re-execution of range-queries, we may get newly added or removed rows.
Moreover, it is the lowest required level to prevent the lost update. The lost update occurs when two or more concurrent transactions read and update the same row. *REPEATABLE_READ* does not allow simultaneous access to a row at all. Hence the lost update can't happen.

```
@Transactional(isolation = Isolation.SERIALIZABLE)
```

SERIALIZABLE is the highest level of isolation. It prevents all mentioned concurrency side effects, but can lead to the lowest concurrent access rate because it executes concurrent calls sequentially.
In other words, concurrent execution of a group of serializable transactions has the same result as executing them in serial.

Isolation Level	dirty read	non-repeatable read	phantom read
READ_UNCOMMITTED	yes	yes	yes
READ_COMMITTED	no	yes	yes
REPEATABLE_READ	no	no	yes
SERIALIZABLE	no	no	no