SPRING DATA TRANSACTION

dasturlash.uz

Data Base Transaction 1

- Transaction is a single unit of work that consists of one or more operations.
- A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing/updating/deleting the contents of the database.
- Tushunmaganlar uchun :
- Transaction bu 1 yoki undan ko’p bo’lgan query larni o’zida jamlagan ish.
- Transaction 1ta user (session) tomonidan bajariladi va baza dagi malumotlarni o’qish/o’zgartirish/ochirish uchun ishlatiladi.

Data Base Transaction 2

- A classical example of a transaction is a bank transfer from one account to another.
- A complete transaction must ensure a balance between the sender and receiver accounts.
- It means that if the sender account transfers X amount, the receiver receives X amount, no more or no less.

Data Base Transaction 3

- The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.
- A PostgreSQL transaction is atomic, consistent, isolated, and durable. These properties are often referred to as ACID:
- The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.
- Transaction ni 4ta xususiyati bor. Ular tranzaktsiyadan oldin va keyin ma'lumotlar bazasida izchillikni saqlash uchun ishlatiladi.
- ACID xususiyati transaction ni boshqarishni tasvirlaydi. ACID Atomicity, Consistency, Isolation va Durability larning qisqartmasidir.

All in Short – Barchasi qisqacha

- **Atomicity** means either all successful or none.
- **Consistency** ensures bringing the database from one consistent state to another consistent state.
- **Isolation** ensures that transaction is isolated from other transaction.
- **Durability** means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Atomicity

- Atomicity guarantees that the transaction completes in an all-or-nothing manner.
- There is no midway, i.e., the transaction cannot occur partially.
- Each transaction is treated as one unit and either run to completion or is not executed at all.
- Atomicity involves the following two operations:
  - **Abort:** If a transaction aborts then all the changes made are not visible.
  - **Commit:** If a transaction commits then all the changes made are visible.
- Atomicity – ato’mga vashe aloqasi yo’q.
- Atomicity tranzaksiyada hammasi yoki hech narsa usulida bajarilishini kafolatlaydi. Yani barchasi bajariladi yoki umuman bajarilmaydi.
- Har bitta Transaksiya bitta birlik deb qaraladi va u yo oxirigacha ishlaydi yoki umuman bajarilmaydi.
- **Atomicity** describes an all or nothing principle

dasturlash.uz

Consistency

- Consistency ensures the change to data written to the database must be valid and follow predefined rules.
- Consistency – doimilik, ketma ketlik,
- Consistency – bazadagi o’zgartitilgan malumotlar haqiqiy bo’lishiga va oldindan belgilangan qoidaga amal qilishini tanimlaydi.

- The **consistency** characteristic ensures that your transaction takes a system from one consistent state to another consistent state. That means that either all operations were rolled back and the data was set back to the state you started with or the changed data passed all consistency checks.
- In a relational database, that means that the modified data needs to pass all constraint checks, like foreign key or unique constraints, defined in your database.

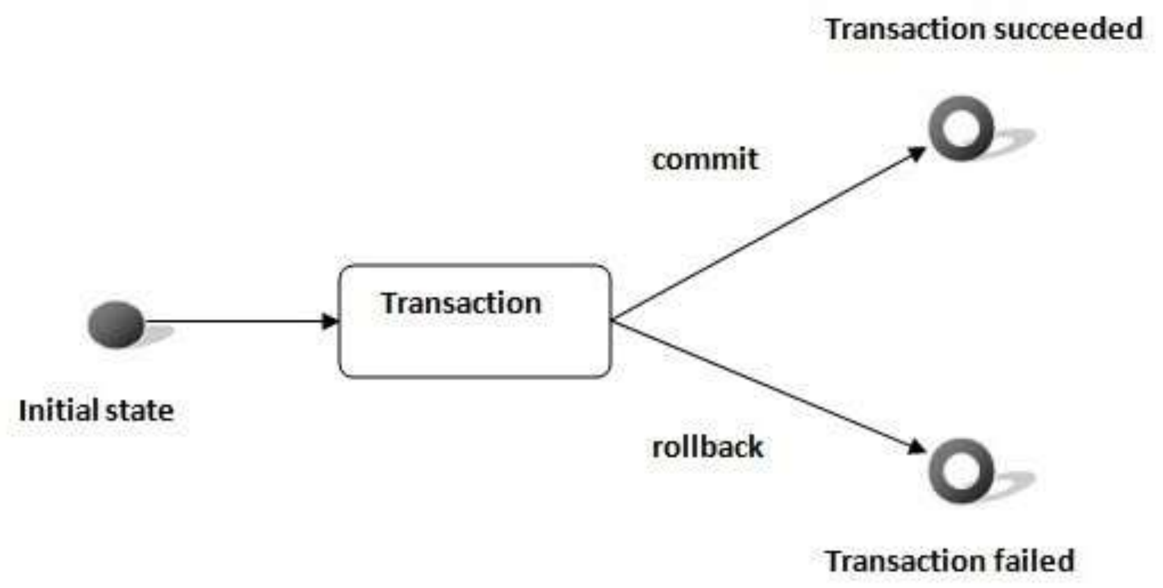
Isolation

- Isolation determines how transaction integrity is visible to other transactions.
- It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
- In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
- The concurrency control subsystem of the DBMS enforced the isolation property.
- Isolation – Izolyatsiya, yakkalab qo'yish, boshqalardan ajratib qo'yish
- Isolation - bu tranzaksiyani boshqa tranzaktsiyalarga qanday ko'rinishini kerakligini aniqlaydi.
- Bu shuni ko'rsatadiki, tranzaktsiyani amalga oshirish vaqtida foydalanilgan ma'lumotlardan birinchisi Transaction tugamaguncha ikkinchi Transaction tomonidan foydalanilmaydi.
- Isolation da : agar T1 tranzaksiyasi bajarilayotgan bo'lsa va X ma'lumotlar elementidan foydalanilsa, T1 tranzaksiyasi tugaguniga qadar ushbu ma'lumotlar elementiga boshqa T2 tranzaktsiyasi orqali kirish mumkin emas.

Durability

- Durability makes sure that transactions that have been committed will be stored in the database permanently.
- **Durability** ensures that your committed changes get persisted.
- Durability – Transaction dagi o’zgargan malumotlar commit qilingandan keyin ular bazada paydo bo’ladi. Yani bazaga tushmasdan qolib ketmaydi.

Transaction in picture



Transaction in database 1

- BEGIN TRANSACTION
- ………..
- COMMIT TRANSACTION;
- Or
- ROLLBACK TRANSACTION;

Transaction in database 2 All example

```
BEGIN;

UPDATE accounts
SET balance = balance - 1000
WHERE id = 1;


UPDATE accounts
SET balance = balance + 1000
WHERE id = 2;

COMMIT;
```

```
BEGIN;

UPDATE accounts
SET balance = balance - 1000
WHERE id = 1;


UPDATE accounts
SET balance = balance + 1000
WHERE id = 2;

ROLLBACK;
```

Advantage of Transaction Mangaement

- **fast performance** It makes the performance fast because database is hit at the time of commit.

JDBC and Transaction

- In JDBC, **Connection interface** provides methods to manage transaction.
- Methods:
- void setAutoCommit(boolean status) - It is true by default means each transaction is committed by default.
- void commit() - commits the transaction.
- void rollback() - cancels the transaction.
-

Simple example of transaction management in jdbc using Statement

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
con.setAutoCommit(false);

Statement stmt=con.createStatement();
stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");

con.commit();
con.close();
```

Example of transaction management in jdbc using PreparedStatement

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
con.setAutoCommit(false);

PreparedStatement ps=con.prepareStatement("insert into user420 values(?,?,?)");

ps.setInt(1,4);
ps.setString(2,"Ali");
ps.setInt(3,3232);
ps.executeUpdate();

System.out.println("commit/rollback");
String answer=br.readLine();
if(answer.equals("commit")){
    con.commit();
  }
if(answer.equals("rollback")){
   con.rollback();
}
```

Using Savepoints 1

- The new JDBC 3.0 Savepoint interface gives you the additional transactional control. Most modern DBMS, support savepoints within their environments such as Oracle's PL/SQ.
- When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.
- Transaction da savePoint qo’yilganda u rollback ishlashi kerak bo’lgan niqtani bildiradi. SavePoint o’rnatilganidan keyin xatolik sodir bo’lsa rollback metodi orqali o’zgarishlarni faqat shu savePoint gacha orqaga qaytarish yoki barcha o’zgarishlarni orqaga qaytarish imkoni bor.

Using Savepoints 2

- The Connection object has two new methods that help you manage savepoints. (SavePoint bilan ishlash uchun Connection ob’ektida 2ta metod bor.)
- **setSavepoint(String savepointName)** − Defines a new savepoint. It also returns a Savepoint object.
- **releaseSavepoint(Savepoint savepointName)** − Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

Using Savepoints 3

- There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.
- rollback(String savePointName) – metodi orqali o’zgarishlarni savePoint gacha rollback qilish imkonini beradi.

Using Savepoints – example

```
try{
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   //set a Savepoint
   Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
   String SQL = "INSERT INTO Employees  VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);


   String SQL = "INSERTED IN Employees VALUES (107, 22, 'Sita', 'Tez')";
   stmt.executeUpdate(SQL);
   conn.commit();

}catch(SQLException se){
   // If there is any error.
   conn.rollback(savepoint1);
}
```

JDBC Transaction Links

- https://www.javatpoint.com/transaction-management-in-jdbc
- https://www.tutorialspoint.com/jdbc/jdbc-transactions.htm

Transactions with Spring

- Spring provides all the boilerplate code that’s required to start, commit, or rollback a transaction.
- It also integrates with Hibernate’s and JPA’s transaction handling.
- Spring Transaction ni start, commit, rolback qilish uchun takrorlanuvchi shablon codelarni o’zi tagdim etadi.
- Shuningdek u Hibernate va JPA da Transaction bilan ishlashni integratsiya qiladi.

Transactions with Spring Boot

- If you’re using Spring Boot, this reduces your effort to a @Transactional annotation on each interface, method, or class that shall be executed within a transactional context.
- Agar siz Spring boot ishlatsangiz , Transaction contentida ishlatiladigan har bir interface, metod yo class ni @Transaction anotatsiyasi bilan belgilashni o’iz kifoya.

@EnableTransactionManagement

- If you’re using Spring without Spring Boot, you need to activate the transaction management by annotating your application class with @EnableTransactionManagement.
- Agar siz Spring ni o’zi Spring boot siz ishlatsangiz Project da Transaction Management yoqish/config qilish uchun @EnableTransactionManagement dan foydalaning.

@Transaction method example 1

- Here you can see a simple example of a service with a transactional method.

```
@Service
public class AuthorService {
    @Autowired
    private AuthorRepository authorRepository;

    @Transactional
    public void updateAuthorNameTransaction() {
        Author author = authorRepository.findById(1L).get();
        author.setName("new name");
    }
}
```

@Transactional Implementation Details 1

- The @Transactional annotation tells Spring that a transaction is required to execute this method.
- When you inject the AuthorService somewhere, Spring generates a proxy object that wraps the AuthorService object and provides the required code to manage the transaction.

- @Transactional annotatsiyasi by metodni ishlatish uchun transaction dan foydalanish kerak ekanligini bildiradi.
- Biz boshqa bitr joyda AuthorService ni inject qilgan paytimizda Spring AuthorService classini o’rab oladigan Proxy ob’ekti yaratadi va transaction uchun kerakli kodlarni taqdim etadi.

@Transactional Implementation Details 2

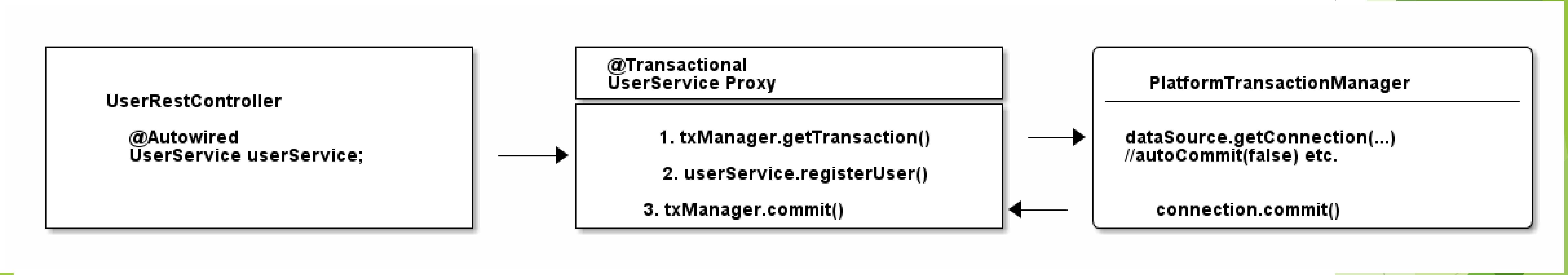
- By default, that proxy starts a transaction before your request enters the first method that's annotated with @Transactional.
- After that method got executed, the proxy either commits the transaction or rolls it back if a RuntimeException or Error occurred.
- Everything that happens in between, including all method calls, gets executed within the context of that transaction.
- Defaul xolatda Proxy ob'ekti transaction ni birinchi @Transction anotatsiyasi bilan belgilangna metod chaqirishdan oldin ishga tushuradi.
- Metod tugaganidan keyin Proxy ob'ekti transaction ni commit qiladi. Agar metodda RuntimeException yoki Error sodir bo'lsa transaction rollback qilinadi.
- O'rtada sodir bo'ladigan barcha narsa, shu jumladan barcha method chaqiruvlari ushbu tranzaksiya kontekstida amalga oshiriladi.

@Transactional Implementation Details 3

- if we have a method like updateAuthorNameTransaction() and we mark it as *@Transactional*, Spring will wrap some transaction management code around the invocation*@Transactional* method called:

```
createTransactionIfNecessary();
try {
    updateAuthorNameTransaction();
    commitTransactionAfterReturning();
} catch (exception) {
    completeTransactionAfterThrowing();
    throw exception;
}
```

@Transactional Implementation Details 4



How to Use *@Transactional 1*

- We can put the annotation on definitions of interfaces, classes, or directly on methods.
- They override each other according to the priority order; from lowest to highest we have: interface, superclass, class, interface method, superclass method, and class method.
- Bu annotatsiyani interface, class yoki metod uchun ishlatsak bo’ladi.
- Ular bir birini xususiyatini pasdan tepaga qarab qayta o’zlashtirishi (override qilishi) mumkin. Yani interface, superclass, class, interface method, superclass method, and class method.

How to Use *@Transactional 2*

- Spring applies the class-level annotation to all public methods of this class that we did not annotate with @Transactional.
- **However, if we put the annotation on a private or protected method, Spring will ignore it without an error.**
- Spring Class ga berilgan annotatsiyani shu class dagi barcha @Transactional yozilmagan public metodlar uchun ham qollaydi.
- Ammo Spring classdagi private yoki protectod metodlarga yozilgan annotatsiyani inkor qiladi.

How to Use *@Transactional* 3

- Usually it's not recommended to set *@Transactional* on the interface; however, it is acceptable for cases like *@Repository* with Spring Data. We can put the annotation on a class definition to override the transaction setting of the interface/superclass:

```
@Transactional
public interface TransferService {
    void transfer(String user1, String user2, double val);
}

@Service
@Transactional
public class TransferServiceImpl implements TransferService {
    @Override
    public void transfer(String user1, String user2, double val) {
        // ...
    }
}
```

@Transactional attributes

- The @Transactional annotation supports a set of attributes that you can use to customize the behavior. The most important ones are propagation, isolation, readOnly, rollbackFor, and noRollbackFor. Let’s take a closer look at each of them.
- @Transaction annotatsiyasi transaction ni xususiyatini/?? o’zgartirish uchun bir nechta atributlarni taqdim etadi. Muhimlari: propagation, isolation, readOnly, rollackFor, noRollbackFor

Read-Only

Using Read-Only Transactions 1

- readOnly – attribute used only executing fetching queries inside class or method.
- If we don’t provide any value for readOnly in @Transactional, then the default value will be false.
- @Transactional(readOnly = true) - used only for retrieval operation to make sure we can only perform the read-only operation.
- readOnly = true – makes shure that metod or class will execute only read operation. Update or Insert Operation not effects any change in DB.
- readOnly xususiyati Class yoki metodda faqat malumot olish uchun query lar ishlatish imkonini beradi.
- Default xolatda @Transactional(readOnly=false) qiymat qo’yilgan.
- @Transactional(readOnly = true) – faqat o’qish uchun query/amal lar bajarish mumkin ekanligini taminlaydi.
- So insert or update can be done inside readOnly method or class.

Using Read-Only Transactions 2

- If we use @Transactional(readOnly = true) to a method which is performing create or update operation then we will not have newly created or updated record into the database but we will have the response data.

```
@Transactional(readOnly = true)
  public void update_balance(String number, Long balance) {
      Optional<CardEntity> optional = cardRepository.findByNumber(number);
      if (optional.isPresent()) {
          CardEntity entity = optional.get();
          entity.setBalance(entity.getBalance() + balance);
          cardRepository.save(entity);
      }
  }
```

Using Read-Only Transactions 2

- We can override readOnly behavior using @Modifying annotation. For example, suppose @Transactional annotation has been used with class level or interface level as below and we want to override readOnly behavior for one method(we don’t want to apply readOnly true for deleteOldBooks() method).

```
@Repository
@Transactional(readOnly = true)
public interface BookRepository extends
CrudRepository<Book,Serializable> {

  List<Book> findByBookName(String bookName);

  @Modifying
  @Transactional
  @Query("delete from Book b where b.old= true")
  void deleteOldBooks();

}
```

Read Only links

- https://thorben-janssen.com/transactions-spring-data-jpa/
- https://www.netsurfingzone.com/spring/transactional-readonly-true-example-in-spring-boot

Handling Exceptions rollbackFor and noRollbackFor

dasturlash.uz

Handling Exceptions 1

- Spring proxy automatically rolls back your transaction if a RuntimeException or Error occurred.
- You can customize that behavior using the rollbackFor and noRollbackFor attributes of the @Transactional annotation.
- Spring proxy ob’ekti Transaction da RuntimeException yoki Error sodir bo’lsa avtomatik ravishda roll back qiladi.
- Bu jarayonni biz @Transactional annotatsiyasining rollbackFor yoki noRollbackFor atributlari orqali o’zgatirishimiz mumkin.

Handling Exceptions 2

- The rollbackFor attribute enables you to provide an array of Exception classes for which the transaction shall be rolled back.
- The noRollbackFor attribute accepts an array of Exception classes that shall not cause a rollback of the transaction.
- rollbackFor attributi Exception array qabul qiladi va shu exception lardan birontasi sodir bo’ls Transaction rollback qilinadi.
- noRollbackFor atributi Exception array qabul qiladi va shu exceptionlar sodir bo’lsa Transction rollback qilinmaydi.

Handling Exceptions Example

- In the following example, the transaction will rollback for all subclasses of the Exception class except the EntityNotFoundException.

```
@Transactional(rollbackFor = Exception.class,  noRollbackFor = EntityNotFoundException.class)
   public void updateAuthorName() {
      Author author = authorRepository.findById(1L).get();
      author.setName("new name");
   }
```

Handling Exceptions Links

- https://thorben-janssen.com/transactions-spring-data-jpa/

Transaction Propagation

Transaction Propagation 1

- Propagation – Tarqatish, tarqalish, ko'paytirish.
- Propagation defines our business logic's transaction boundary.
- Spring manages to start and pause a transaction according to our *propagation* setting.
- Spring calls *TransactionManager::getTransaction* to get or create a transaction according to the propagation.
- It supports some of the propagations for all types of *TransactionManager*, but there are a few of them that are only supported by specific implementations of *TransactionManager*.
- Propagation – Transaksiyani biznes logikasini chegarasini ko’rsatadi.
- Spring propagation xususiyayini qiymatiga qarab Transaction ni yaratish yoki to’xtatib turish kerakligini aniqlab oladi.
- Spring propagation ga qarab yangi transaction yaratish uchun *TransactionManager::getTransaction* metodini chaqiradi.
- U bazibir Porpagation qiymatini barcha TransactionManager lar uchun qo’llab quvvatlaydi. Ammo bazir bir qiymatlar faqat malum bir TransactionManager lar tomonidan ishlatilishi mumkin.

dasturlash.uz

Transaction Propagation 2

- Propagation property values
  - *REQUIRES_NEW* Propagation
  - *REQUIRED* Propagation
  - *NESTED* Propagation
  - *MANDATORY* Propagation
  - *SUPPORTS* Propagation
  - *NOT_SUPPORTED* Propagation
  - *NEVER* Propagation

Transaction Propagation 2

- REQUIRES_NEW - Always executes in a new transaction. If there is any existing transaction it gets suspended
- REQUIRED - Always executes in a transaction. If there is any existing transaction it uses it. If none exists then only a new one is created.
- NESTED - Always executes in a transaction. If a transaction exists, it marks a save point. If there's no active transaction, it works like *REQUIRED*.
- MANDATORY - Always executes in a transaction. If there is any existing transaction it is used. If there is no existing transaction it will throw an exception.
- SUPPORTS - It may or may not run in a transaction. If current transaction exists then it is supported. If none exists then gets executed with out transaction.
- NOT_SUPPORTED - Always executes without a transaction. If there is any existing transaction it gets suspended.
- NEVER - Always executes with out any transaction. It throws an exception if there is an existing transaction

REQUIRES_NEW Propagation

- When the propagation is *REQUIRES_NEW*, Spring suspends the current transaction if it exists, and then creates a new one:
- REQUIRED_NEW da Spring hozirgi active Transaction ni to’xtatadi va yangi Transaction yaratadi. Yangi yaratilgan Transactionda metod amalga oshiriladi.

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void requiresNewExample(String user) {
    // ...
}
```

The pseudo-code looks like so:

```
if (isExistingTransaction()) {
    suspend(existing);
    try {
        return createNewTransaction();
    } catch (exception) {
        resumeAfterBeginException();
        throw exception;
    }
}
return createNewTransaction();
```

REQUIRED Propagation 1

- *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:
- Required bu default propagation dir. Agar Transaction yaratilmagan bo’lsa Spring Trasction yaratadi. Agar Transaction mavjut bo’lsa Spring shu transaction ni ishlatadi.

```
@Transactional(propagation = Propagation.REQUIRED)
public void requiredExample(String user) {
    // ...
}
```

REQUIRED Propagation 2

- Furthermore, since *REQUIRED* is the default propagation, we can simplify the code by dropping it:
- Bundan tashqari REQUIRED defaul propagation bo’lgani uchun uni yozib o’tirish shart emas. Mazgi.

```
@Transactional
public void requiredExample(String user) {
    // ...
}
```

REQUIRED Propagation 3

- Let's see the pseudo-code of how transaction creation works for *REQUIRED* propagation:

```
if (isExistingTransaction()) {
    if (isValidateExistingTransaction()) {
        validateExisitingAndThrowExceptionIfNotValid();
    }
    return existing;
}
return createNewTransaction();
```

NESTED Propagation

- 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*.
- NESTED da Spring active transaction bormi yo’qmi tekshiradi. Agar mavjut bo’lsa shu to’chkada savePoint yaratadi. Agar Xatalik bo’lsa Transaction save pointgacha rollback qilinadi. Agar active Transaction bo’lmasa bu xuddi REQUIRED kabi ishlaydi.

- ***JpaTransactionManager*** **supports** ***NESTED*** **only for JDBC connections. However, if we set the** ***nestedTransactionAllowed*** **flag to** ***true*****, it also works for JDBC access code in JPA transactions if our JDBC driver supports save points.**

NESTED Propagation 1

```
@Transactional(propagation = Propagation.NESTED)
public void nestedExample(String user) {
    // ...
}
```

MANDATORY Propagation

- 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:

- MANDATORY da active bo’lgan Transction bo’lsa shu ishlatiladi. Agar active transaction bo’lmasa Spring Exception tashaydi.

```
@Transactional(propagation = Propagation.MANDATORY)
public void mandatoryExample(String user) {
    // ...
}
```

MANDATORY - Let's again see the pseudo-code:

```
if (isExistingTransaction()) {
   if (isValidateExistingTransaction()) {
       validateExisitingAndThrowExceptionIfNotValid();
     }
  return existing;
}

throw IllegalTransactionStateException;
```

SUPPORTS Propagation 1

- 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:
- SUPPORTS qiymatida Spring oldin active Transction borligini tekshiradi. Agar Trasaction bo’lsa shuni ishlatadi. Agar Transaction yo’q bo’lsa transacsiyasiz query larni amalga oshiradi (metodni transaction siz bajaradi).

```
@Transactional(propagation = Propagation.SUPPORTS)
public void supportsExample(String user) {
    // ...
}
```

SUPPORTS Propagation 2

- Let's see the transaction creation's pseudo-code for *SUPPORTS*:

```
if (isExistingTransaction()) {
    if (isValidateExistingTransaction()) {
        validateExisitingAndThrowExceptionIfNotValid();
    }
    return existing;
}
return emptyTransaction;
```

NOT_SUPPORTED Propagation

- If a current transaction exists, first Spring suspends it, and then the business logic is executed without a transaction.
- Agar Active Transaction bo’lsa uni to’xtatib turib biznez logikani (metodni) Transaction siz amalga oshiradi.

```
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public void notSupportedExample(String user) {
    // ...
}
```

NEVER Propagation 1

- For transactional logic with *NEVER* propagation, Spring throws an exception if there's an active transaction:
- NEVER qiymatda agar active Transaction bo’lsa Exception tashaydi.

```
@Transactional(propagation = Propagation.NEVER)
public void neverExample(String user) {
    // ...
}
```

NEVER Propagation 2

- Let's see the pseudo-code of how transaction creation works for *NEVER* propagation:
-

```
if (isExistingTransaction()) {
    throw IllegalTransactionStateException;
}
return emptyTransaction;
```

NEVER Propagation 2 – Ishlamadimi?

- Transactions in Spring are proxy-based: when a bean A calls a transactional bean B, it actually calls a method of a dynamic proxy, which deals with the opening of the transaction, then delegates to the actual bean B, then deals with the commit/rollback of the transaction.
- If you call a method2 from a method1 of a single bean A, your call is not intercepted by the transactional proxy anymore, and Spring is thus completely unaware that method2() has been called. So nothing can check that there is no transaction.
- Put the method2 in *another* bean, injected in BookManager, and everything will work as expected.
-

Propagation Links

- https://www.baeldung.com/spring-transactional-propagation-isolation
- https://www.techgeeknext.com/spring-boot/spring-boot-transaction-propagation

Transaction Isolation

Transaction Isolation – definition 1

- Isolation – Izolyatsiya, yakkalab qo'yish, boshqalardan ajratib qo'yish

- 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.
- Isolation bu ACID ning 4ta xususiyatidan bittasi dir.
- Isolation bu bir vaqtni o'zida bajarilgan Transaction larning qilgan o'zgarishlari bir biriga qachon ko'rinishi kerak ekanligini aniqlaydi.

Transaction Isolation – definition 2

- 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.

- Isolation level da bitta Transaction da o’zgartirilgan malumotlar shu vaqtni o’zida bajarilgan boshqa bir Transaction ga tasirini bildiradi. Yani qachon va qanday qilib o’zgarishlar boshqa bir Transaction da ko’rinishi kerakligini ko’rsatadi.
- Spring framework yordamida Transaction yaratsak uning uning isolation level lini ham ko’rsatib ketsak bo’ladi.

Transaction Isolation - values

- READ_UNCOMMITTED
- READ_COMMITTED
- REPEATABLE_READ
- SERIALIZABLE

Usage example

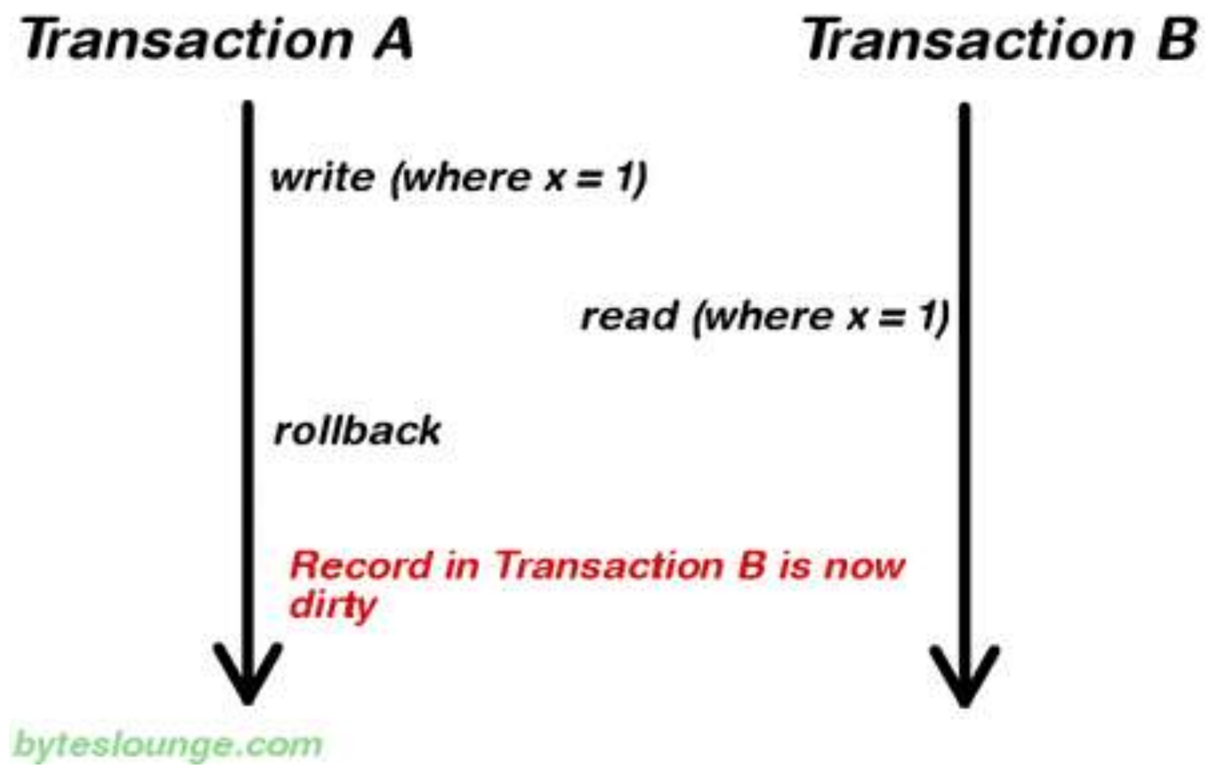
- Using the **@Transactional** annotation we can define the isolation level of a Spring managed bean transactional method. This means that the transaction in which this method is executed will run with that isolation level:

```
@Transactional(isolation=Isolation.READ_COMMITTED)
public void someTransactionalMethod(User user) {

  // Interact with testDAO

}
```

Isolation **READ_UNCOMMITTED**

- **READ_UNCOMMITTED** isolation level states that a transaction **may** read data that is still **uncommitted** by other transactions.
- **READ_UNCOMMITTED da bitta Transaction dagi commit bo'lmagan o'zgarishlar ikkinchia Transaction da ham ko'rinadi.**
- This constraint is very relaxed in what matters to transactional concurrency but it may lead to some issues like dirty reads.
- Bu levelda Parallel bo'ladigan Transaction larda dirty reads muommosiga olib keladi.

Isolation **READ_UNCOMMITTED**

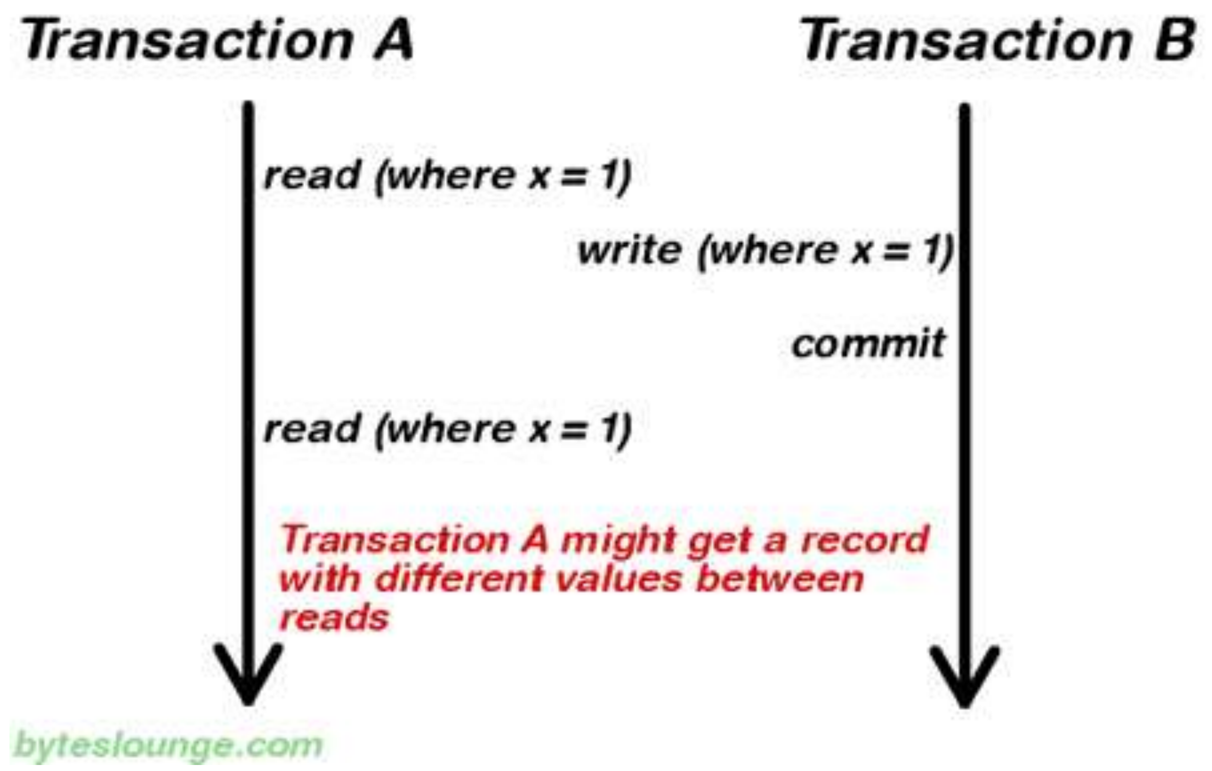


- In this example **Transaction A** writes a record. Meanwhile **Transaction B** reads that same record before **Transaction A** commits. Later **Transaction A** decides to rollback and now we have changes in **Transaction B** that are inconsistent. This is a **dirty read**. **Transaction B** was running in **READ_UNCOMMITTED** isolation level so it was able to read **Transaction A** changes before a commit occurred.

Isolation READ_COMMITTED 1

- **READ_COMMITTED** isolation level states that 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.
- READ_COMMITTED level da Transctionda sodir bo'lgan o'zgarishlar hali commit bo'lmasidan turib boshqa Transaction ularni ko'ra olmaydi.
- Bunda dirty read muommosi sodir bo'lmaydi. Ammo **non-repeatable read** muommosi sodir bo'lishi mumkin.

Isolation READ_COMMITTED 2



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

Isolation REPEATABLE_READ 1

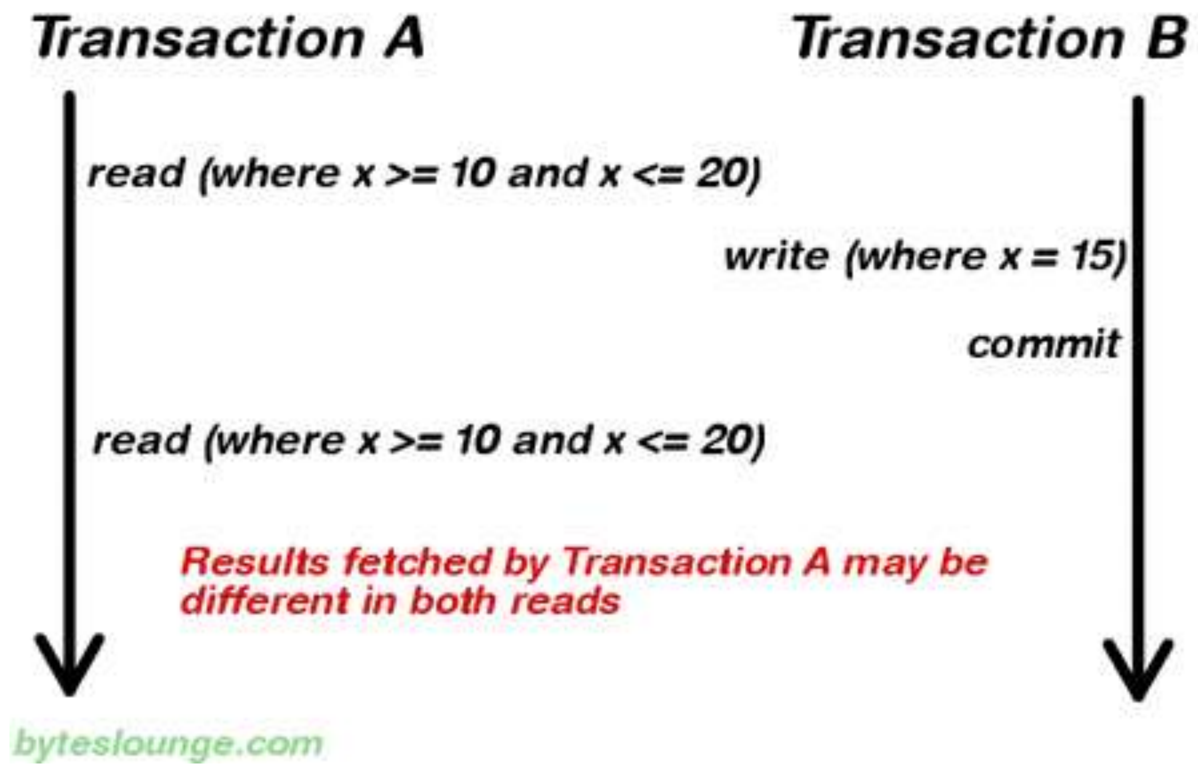
- If two transactions are executing concurrently - **till the first transaction is committed the existing records cannot be changed by second transaction but new records can be added.**

- Ikkita Transction bir vaqtni o’zida ishlasa – bor bo’lgan malumotlar birinchi transaction da commit bo’lmasidan oldin ularni Ikkinchi Transaction da o’zgartirib bo’lmaydi. Ammo Yangi qo’shilgan Malumotlar ko’rinadi.
- Dehqonchasiga Bor bo’lgan malumotlarni o’zgartirsa ular boshqa transaction da ko’rinmaydi. Ammo yangi malumot qo’shsa ular boshqa Transactionda ko’rinadi.

Isolation REPEATABLE_READ 2

- **REPEATABLE_READ** isolation level states that 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.
- **REPEATABLE_READ** levelda Transaction bitta query orqali malumotlarni bir necha marta o'qisa ham malumotlar har doim birxil bo'ladi.
- Bu **dirty read** va **no-repeatable** muommolarni bartaraf qiladi, Ammo boshqa **phantom read** muommo paydo bo'lishi mumkin.

Isolation REPEATABLE_READ 3


Transaction A
Transaction B
read (where x >= 10 and x <= 20)
write (where x = 15)
commit
read (where x >= 10 and x <= 20)
Results fetched by Transaction A may be different in both reads
byteslounge.com

- In this example **Transaction A** reads a **range** of records. Meanwhile **Transaction B** inserts a new record in the same range that **Transaction A** initially fetched and commits. Later **Transaction A** reads the same range again and will also get the record that **Transaction B** 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).

dasturlash.uz

Isolation **SERIALIZABLE**

- If two transactions are executing concurrently then it is as if the transactions get executed serially i.e the first transaction gets committed only then the second transaction gets executed.
- This is **total isolation**. So a running transaction is never affected by other transactions. However this may cause issues as **performance will be low and deadlock might occur**.
- Agar ikkita tranzaktsiya bir vaqtning o'zida amalga oshirilsa, u holda tranzaktsiyalar ketma-ket bajarilgandek bo'ladi, ya'ni birinchi tranzaktsiya amalga oshirilgandan keyingina ikkinchi tranzaksiya amalga oshiriladi.
- SERIALIZABLE to'liq izolyatsiya dir. Shunday qilib, ishlaydigan tranzaktsiyaga boshqa tranzaktsiyalar hech qachon ta'sir qilmaydi. Biroq, bu muammoga olib kelishi mumkin, chunki unumdorlik past bo'ladi va tiqilib qolishi mumkin.

Isolation - DEFAULT

- DEFAULT isolation level, as the name states, uses the default isolation level of the datastore we are actually connecting from our application.
- DEFAULT da ishlayotgan bazaning Isolation levveli tanlab ishlatiladi.

Isolation - Links

- https://www.byteslounge.com/tutorials/spring-transaction-isolation-tutorial
- https://www.javainuse.com/spring/boot-transaction-isolation
- https://habr.com/ru/post/513644/
- https://www.baeldung.com/spring-transactional-propagation-isolation

Isolation – Problems: **Dirty Reads**

- **Dirty Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A modifies a record but not commits it. Transaction B reads this record but then Transaction A again rollbacks the changes for the record and commits it. So Transaction B has a wrong value
- Dirty reads – tasavvur qiling sizda 2ta bir vaqtni o’zida ishlaydigan A va B Transaction lar bor. Transaction A malumotni o’zgartirda lekin uni hali commit qilmadi. Shu paytda Transaction B o’zgargan malumotni o’qib oldi. Ammo Transaction A o’zgartirgan malumotni rollback qildi va commit qildi. Natijada Transaction B da noto’gri malumot bor.

Isolation – Problems: **Non-Repeatable Reads**

- **Non-Repeatable Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A reads some records. Transaction B modifies these records before transaction A has been committed. So if Transaction A again reads these records they will be different. So same select statements result in different existing records.
- **Non-Repeatable Reads –** Transaction A va Transaction B bir vaqtni o’zida ishga tushdi. Transaction A malumotni o’qib oldi o’zgartirishin boshladi. Transaction B shu malumotni o’qib olib, o’zgartirib uni commit qildi. Transaction A yana malumotni o’qib olsa unga boshqacha malumot keladi. Yani birxil select query boshqacha malumotni jo’natadi.

Isolation – Problems: **Phantom Reads**

- **Phantom Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A reads some records. Transaction B adds more such records before transaction A has been committed. So if Transaction A again reads there will be more records than the previous select statement. So same select statements result in different number records to be displayed as new records also get added.
- Phantom Reads - Transaction A va Transaction B bir vaqtni o’zida ishga tushdi. Transaction A malumotlarni o’qib oldi. Transaction A commit qilmasidan oldin Transaction B yangi malumotlar qo’shdi. Transaction A yana malumotlarni o’qiganida birinchi o’qigan malumotlaridan ko’p malumot bo’ladi. Yani bitta select query turli malumotlar hajmini return qiladi.

Isolation - Summary

- To summarize, the existing relationship between isolation level and read phenomena may be expressed in the following table:
- Xulosa qilib aytganda, izolyatsiya darajasi va o'qish hodisalari o'rtasidagi mavjud bog'liqlik quyidagi jadvalda ifodalanishi mumkin:

	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 Transaction - Links

- https://www.javainuse.com/spring/boot-transaction
- https://thorben-janssen.com/transactions-spring-data-jpa/
- https://www.marcobehler.com/guides/spring-transaction-management-transactional-in-depth#_fin