13. Spring boot @Transactional Annotation

Spring boot @Transactional Annotation - Part1

Before starting @Trasactional in Spring boot, I would highly recommended to watch:

- Concurrency control Video to understand Transaction in depth **HLD** Play List.
- And Spring boot **AOP**

Critical Section

- code segment, where shared resources are being accessed and modified.

Car DB

ID	Status
1001	Available

```
{
   Read Car Row with id:1001
   if Status is Available:
      Update it to booked.
}
```

When multiple request try to access this critical section, Data Inconsistency can happen.

- Let's we can say 4 people try to access car booking critical section in parallel. they all 4 got booking.

Its solution is usages of TRANSACTION

It helps to achieve ACID property

A[Atomicity]:

- Ensures all operations within a transaction are completed successfully. if any operation fails, the entire transaction will get rollback.

C[Consistency]:

- Ensures that DB state before and after the transactions should be consistent only.
- If debit happens for 5 rupees it should be credited for another respective registered user account, Maintaining Consistency.

I[Isolation]

- Ensures that, even if multiple transactions are running in parallel, they do not interfere with each other.
- If multiple transaction running in parallel, but all transaction should perform operation individually without interfering each other.
- They should fell they are isolated, or Independent of each other.
- If there is a three transaction t1,t2 and t3. and they all want to update a critical section called **A.**
- let's t2 got the first chance to access critical section A, then t2 put lock over it, until t2 execution completion, rest of transaction can not access the critical section A. they need to wait for execution completed.
- even though from seeing outside world transaction are running in parallel , but it's not they are following proper sequence.
- Internally transaction make use of locking and make sure they are following proper sequence.

D[Durability]:

- Ensures that committed transaction will never lost despite system failure crash.

TRANSACTION ARE VERY POPULAR IN FINANCIAL SECTOR

BEGIN_TRANSACTION:

- Debit from A

- Credit to B ---ACTUAL BUSINESS LOGIC

If all success:

COMMIT;

```
ELSE
   ROLLBACK;
END_TRANSACTION;
```

If there id 1000 of class or methods then we need to put for each & every class or methods.

SOLUTION:

In Spring boot, we can use **@Transactional** annotation

1.We need to add below Dependency in pom.xml

- Based on DB we are using, suppose we are using RELATIONAL DB

Spring boot DATA JPA **[Java persistence API]**

- helps to interact with RELATIONAL DB without

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>3.3.3</version>
</dependency>
```

Database driver dependency is also required [that we will see in next topic]

2. Activate, Transaction Management by using @EnableTransactionManagement in main class.

- Spring boot generally Auto configure it , so we don't need to specially add it.

```
@SpringBootApplication
@EnableTransactionManagement
public class SpringCourseProjectApplication {
    public static void main(String[] args) {
       SpringApplication.run(SpringCourseProjectApplication.class, args);
    }
}
```

@Transactional

1. **At class level**

- All public method will get execute in transaction context.

- but private method cannot be.

```
package com.springProject.SpringCourseProject.transactional;

import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

@Component
@Transactional
public class ClassLevelCarService {

    public void updateCar(){
        //this method will get executed within a transaction
    }

    public void updateBulkCar(){
        //this method will get executed within a transaction
    }

    private void helperMethod(){
        //this method will not get executed within a transaction
    }
}
```

2. **At Method level**

- Only method annotated with @Transanctional will get executed.
- @Transactional apply on particular method

```
package com.springProject.SpringCourseProject.transactional;

import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

@Component
public class MethodLevelCarServices {

    @Transactional
    public void updateCar(){
        //this method will get executed within a transaction
    }

    public void updateBulkCar(){
        //this method will not get executed within a transaction
    }
}
```

Transaction Management in Spring boot uses AOP.

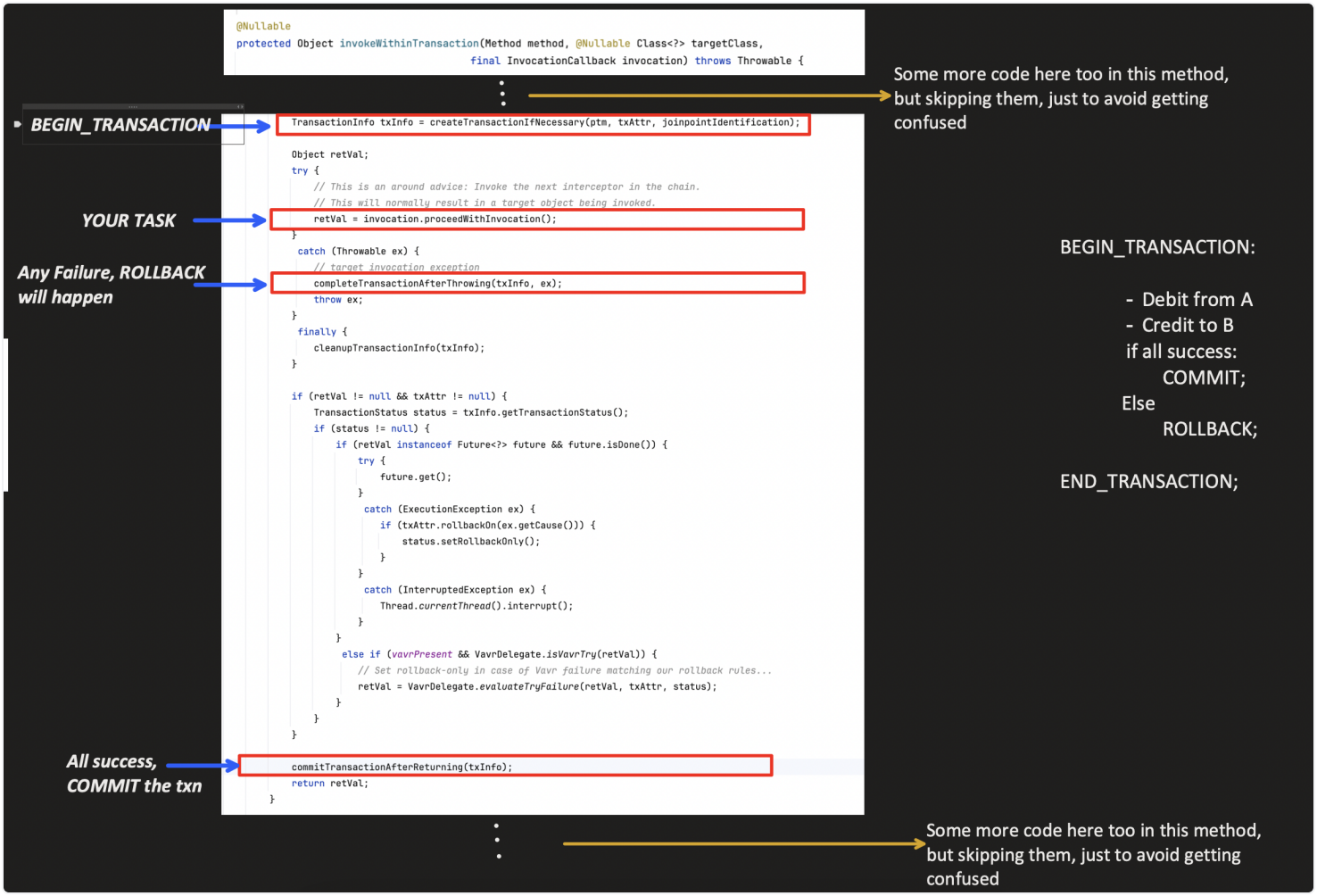
- Uses Point cut expression to search for method, which has @Transactional annotation like:
  >> @within(org.springframework.transaction.annotation.Transctional
- Once Point cut expression matches run an @Around type Advice.

>>> **invokeWithinTransaction** method present in **TransactionalInterceptor** class.

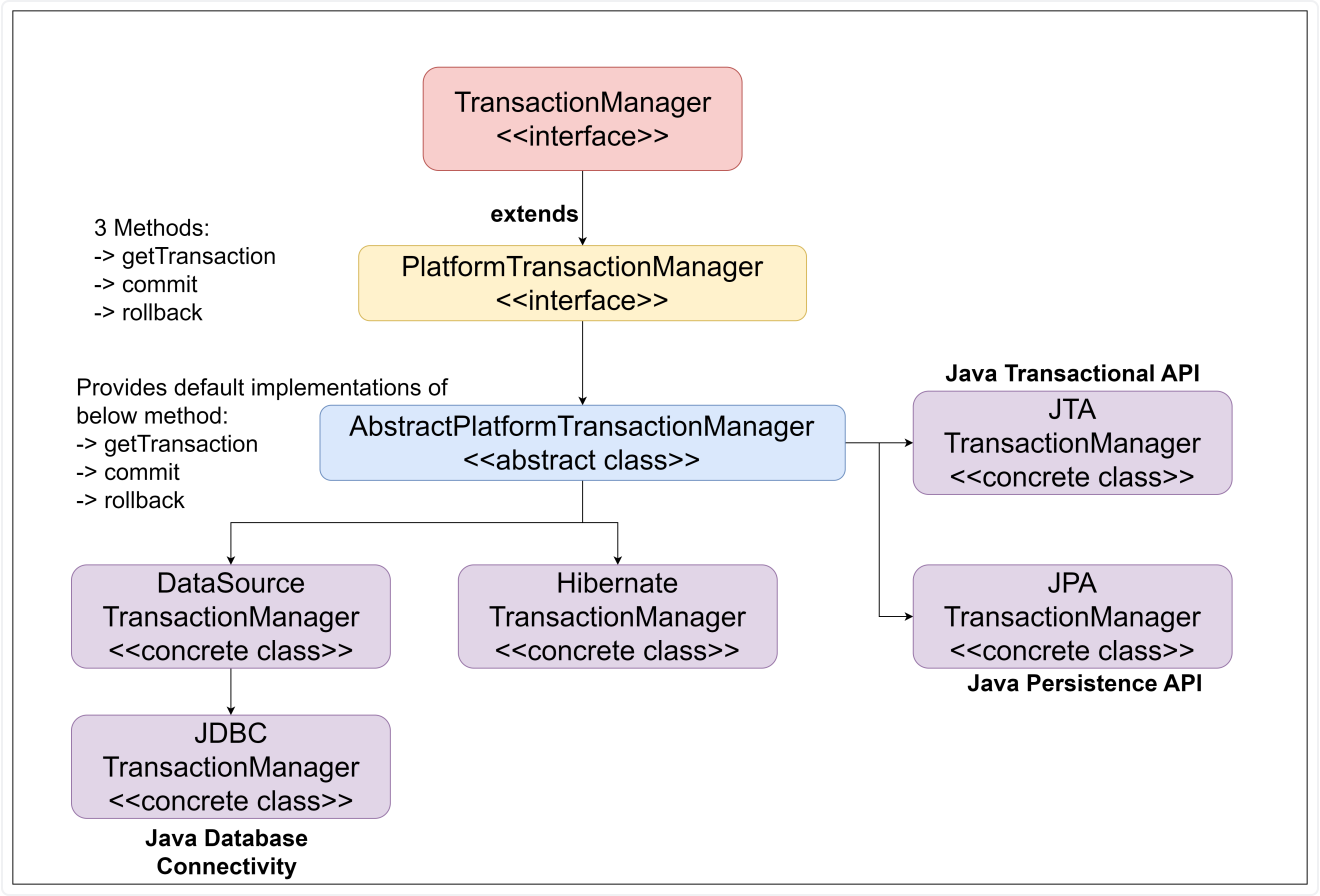
```
package com.springProject.SpringCourseProject.controller;

import com.springProject.SpringCourseProject.Services.UserServices;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/api/")
public class UserController {


    @Autowired
    UserServices userServices;

    @GetMapping(path = "/updateUser")
    public String updateUser(){
        System.out.println("Inside updateUser() method of UserController.");
        userServices.updateUser();
        System.out.println("User Updated Successful.");
        return "User Updated Successful.";
    }
}
```

```
package com.springProject.SpringCourseProject.Services;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class UserServices {

    @Transactional
    public void updateUser(){
```

```
        System.out.println("Inside UserServices class method name updateUser()");
        System.out.println("UPDATE QUERY TO update the user db values");
    }
}
```



Spring boot @Transactional Annotation - Part2 | Declarative, Programmatic Approach and Propagation



- These All Manage LOCAL Transactions
- JTA Manage Distributed Transactions. [2 - PHASE COMMIT]

Types Of Transaction Management.

1. **Declarative** - Transaction Management through Annotation.

```
package com.springProject.SpringCourseProject.Services;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class UserServices {

    @Transactional
    public void updateUser(){
        System.out.println("Inside UserServices class method name updateUser()");
        System.out.println("UPDATE QUERY TO update the user db values");
    }
}
```

- Here, base on underlyin DataSource used like JDBC or JPA etc.
- Spring boot will choose appropirate Transaction manager.
- if i am using @Tranasactional but i want to tell spring boot choose the Given Transactiona Manager By me like specifically use whether it is hibernate , JPA, JDBC and JTA.
- That's why AppConfig Comes into the picture.

```
package com.springProject.SpringCourseProject.config;

import com.springProject.SpringCourseProject.component.Order;
import com.springProject.SpringCourseProject.imple.OfflineOrder;
import com.springProject.SpringCourseProject.imple.OnlineOrder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.transaction.PlatformTransactionManager;

import javax.sql.DataSource;

@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUsername("root");
        dataSource.setPassword("root");

        return dataSource;
    }

    @Bean
    public PlatformTransactionManager userTransactionManager(DataSource dataSource){
        return new DataSourceTransactionManager(dataSource);
    }
}
```

```
package com.springProject.SpringCourseProject.transactional.transactionManagerTypes;

import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;
```

```
@Component
public class UserDeclarative {

    @Transactional(transactionManager = "userTransactionManager")
    public void updateUserProgrammatic(){
        //SOME DB OPERATIONS
        System.out.println("Insert Query ran");
        System.out.println("Update Query ran");
    }
}
```

2. Programmatic

- Transaction Management through code
- Flexible but difficult to maintain.

Why Flexible ? let's consider the below code.

```
package
com.springProject.SpringCourseProject.transactional.transactionManagerTypes.programmatic
;

import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

@Component
public class UserProgrammatic {

    @Transactional
    public void updateUser(){
        //1. update DB initial db operation
        //2. External API CALL
        //3. Update DB final db operation
    }
}
```

- Here We have 3 - operations
- now we are @Transactional annotaion on top.
- it will cause an issue.

- because here external API CALLED IS INVOLVED.
- like third party integeration and it will take 3-4sec to return the response
- so it lie between intial and final db operation so it's hold 3 to 4 sec current db connection
- At the time of peak traffic where you have to open large number db connection.
- this external API CAll will be bottle neck.
- external api calls are time taking and that amount of time DB connection will be on hold
- simply you can not use here @Transactional because it will chocke your system.

Solution Programmatic will help.

```
package com.springProject.SpringCourseProject.config;

import com.springProject.SpringCourseProject.component.Order;
import com.springProject.SpringCourseProject.imple.OfflineOrder;
import com.springProject.SpringCourseProject.imple.OnlineOrder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.transaction.PlatformTransactionManager;

import javax.sql.DataSource;

@Configuration
public class AppConfig {
    @Bean
    public DataSource dataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUsername("root");
        dataSource.setPassword("root");

        return dataSource;
    }

    @Bean
    public PlatformTransactionManager userTransactionManager(DataSource dataSource){
        return new DataSourceTransactionManager(dataSource);
    }
}
```

```
package
com.springProject.SpringCourseProject.transactional.transactionManagerTypes.programmatic
;

import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;

@Component
public class UserProgrammaticApproach1 {
    //this bean will take from Appconfig
    PlatformTransactionManager userTransactionManager;

    UserProgrammaticApproach1(PlatformTransactionManager userTransactionManager){
        this.userTransactionManager=userTransactionManager;
    }

    public void updateUserProgrammatic(){
        TransactionStatus status = userTransactionManager.getTransaction(null);

        try{
            //SOME INITIAL SET OF OPERATIONS
            System.out.println("Insert Query run1");
            System.out.println("Update Query run1");
            userTransactionManager.commit(status);
        }catch (Exception e){
            userTransactionManager.rollback(status);
        }
    }
}
```

Approach 2 Usage of Transaction Template cleaner approach

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

package org.springframework.transaction.support;

import org.springframework.lang.Nullable;
import org.springframework.transaction.TransactionStatus;

@FunctionalInterface
public interface TransactionCallback<T> {
    @Nullable
```

```
    T doInTransaction(TransactionStatus status);
}
```

```
//this tis Part TransactionTemplate class code
@Override
@Nullable
public <T> T execute(TransactionCallback<T> action) throws TransactionException {
    Assert.state(this.transactionManager != null, "No PlatformTransactionManager set");
    PlatformTransactionManager var3 = this.transactionManager;
    if (var3 instanceof CallbackPreferringPlatformTransactionManager cpptm) {
        return cpptm.execute(this, action);
    } else {
        //here i am giving the specific transctional manager by creating bean which is
DataSource
        TransactionStatus status = this.transactionManager.getTransaction(this);

        Object result;
        try {
            //ultimately they are calling our business logic our functional interface
implementation
            result = action.doInTransaction(status);
        } catch (Error | RuntimeException var6) {
            Throwable ex = var6;
            this.rollbackOnException(status, ex);
            throw ex;
        } catch (Throwable var7) {
            Throwable ex = var7;
            this.rollbackOnException(status, ex);
            throw new UndeclaredThrowableException(ex, "TransactionCallback threw
undeclared checked exception");
        }

        this.transactionManager.commit(status);
        return result;
    }
}
```

```
package com.springProject.SpringCourseProject.config;

import com.springProject.SpringCourseProject.component.Order;
import com.springProject.SpringCourseProject.imple.OfflineOrder;
import com.springProject.SpringCourseProject.imple.OnlineOrder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
```

```
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.support.TransactionTemplate;

import javax.sql.DataSource;

@Configuration
public class AppConfig {
    @Bean
    public Order createOrderBean(@Value("${isOnlineOrder}") boolean isOnlineOrder){
        return isOnlineOrder ? new OnlineOrder():new OfflineOrder();
    }


    @Bean
    public DataSource dataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUsername("root");
        dataSource.setPassword("root");

        return dataSource;
    }

    @Bean
    public PlatformTransactionManager userTransactionManager(DataSource dataSource){
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean
    public TransactionTemplate transactionalTemplate(PlatformTransactionManager
userTransactionManager2){
        return new TransactionTemplate(userTransactionManager2);
    }
}
```

```
package
com.springProject.SpringCourseProject.transactional.transactionManagerTypes.programmatic
;

import org.springframework.stereotype.Component;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;

@Component
public class UserProgrammaticApproach2 {

    TransactionTemplate transactionTemplate;
```

```
    public UserProgrammaticApproach2(TransactionTemplate transactionTemplate){
        this.transactionTemplate=transactionTemplate;
    }

    //TransactionCallback<TransactionStatus> functional interface
    public void updateUserProgrammatic(){
        TransactionCallback<TransactionStatus> dbOperationTask =(TransactionStatus
status) ->{
            System.out.println("Insert Query Ran");
            System.out.println("Update Query Ran");
            return status;
        };

    TransactionStatus status = transactionTemplate.execute(dbOperationTask);



    }

}
```

Now, Let's see Propagation

- When we try to create a new Transaction, it first check the **PROPAGATION** value set, and this tell whether have to create new transaction or not.
- Based On Propagation value transaction will be created or not be.

>Suppose you have already a transaction on some method which is doing something.

>Method1 internally call method2() is also as @Transaction;

>Internally method 2 is also trying to getTransaction.

>So question is that is method 2 be part of method 1 transaction or method 2 create will another transaction.

That's where **PROPAGATION** Comes into the picture.

1. **REQUIRED [DEFAULT PROPAGATION]**

@Transactional (prpoagation=Propoagation.REQUIRED)

- if [parent txn prsent]
  use it.

else

CREATE NEW transaction.

```
package com.springProject.SpringCourseProject.DAO;

import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.transaction.support.TransactionSynchronizationManager;

@Component
public class UserDao {


    @Transactional(propagation= Propagation.REQUIRED)
    public void method2(){
        boolean isTransactionActive =
TransactionSynchronizationManager.isActualTransactionActive();
        String currentTransactionName =
TransactionSynchronizationManager.getCurrentTransactionName();

        System.out.println("************ IN METHOD2 OF USER DAO START**************");

        System.out.println("Propagation.REQUIRED_NEW: Is Parent transaction active:
"+isTransactionActive );
        System.out.println("Propagation.REQUIRED_NEW: Is Parent transaction name:
"+currentTransactionName);


        System.out.println("************ IN METHOD2 OF USER DAO END**************");

    }
}
```

```
package com.springProject.SpringCourseProject.Services;

import com.springProject.SpringCourseProject.DAO.UserDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.transaction.support.TransactionSynchronizationManager;
```

```
@Service
public class UserServices {

    @Autowired
    UserDao userDao;



    @Transactional
    public void method1(){

        System.out.println("IN METHOD1 of UserService IS Transaction active: "+
TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("IN METHOD1 IS Transaction name: "+
TransactionSynchronizationManager.getCurrentTransactionName());
        userDao.method2();
        System.out.println("IN METHOD1 of UserService Some initial DB OPERATION");
        System.out.println("IN METHOD1 of UserService Some Final DB OPERATION");
    }

    public void updateUserFromNonTransactional(){

    }
}
```

```
package com.springProject.SpringCourseProject.controller;

import com.springProject.SpringCourseProject.Services.UserServices;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/api/")
public class UserController {



    @Autowired
    UserServices userServices;

    @GetMapping(path = "/updateUser/Propagation_Required_Example")
    public String Propagation_Required_Example(){
        System.out.println("Inside Propagation_Required_Example() method of
UserController.");
```

```
            userServices.method1();
            userServices.updateUserFromNonTransactional();
            return "Propagation_Required_Example() method completed";
        }
    }
```

OUTPUT Hitting API:- http://localhost:8080/api/updateUser/Propagation_Required_Example

Inside Propagation_Required_Example() method of UserController.

IN METHOD1 of UserService IS Transaction active: **true**

IN METHOD1 IS Transaction name: **com.springbootcourse.userservice.method1**

********** IN METHOD2 OF USER DAO START**********

Propagation.REQUIRED_NEW: Is Parent transaction active: **true**

Propagation.REQUIRED_NEW: Is Parent transaction **name: com.springbootcourse.userservice.method1**

********** IN METHOD2 OF USER DAO END**********

IN METHOD1 of UserService Some initial DB OPERATION

IN METHOD1 of UserService Some Final DB OPERATION

2. REQUIRED_NEW:

@Transactional(propagation=Propagation.REQUIRED_NEW)

IF [PARENT TXN PRESENT]

Suspend the parent txn;

Create a new Txn and once finished;

Resume the parent txn;

else

Create new transaction and execute the method

```
  @Transactional(propagation= Propagation.REQUIRED_NEW)
    public void method2(){
        boolean isTransactionActive =
TransactionSynchronizationManager.isActualTransactionActive();
        String currentTransactionName =
TransactionSynchronizationManager.getCurrentTransactionName();

        System.out.println("************ IN METHOD2 OF USER DAO START**************");

        System.out.println("Propagation.REQUIRED_NEW: Is Parent transaction active:
"+isTransactionActive );
        System.out.println("Propagation.REQUIRED_NEW: Is Parent transaction name:
"+currentTransactionName);


        System.out.println("************ IN METHOD2 OF USER DAO END**************");

    }
```

OUTPUT Hitting API:- http://localhost:8080/api/updateUser/Propagation_Required_Example

Inside Propagation_Required_Example() method of UserController.

IN METHOD1 of UserService IS Transaction active: true

IN METHOD1 IS Transaction name: com.springbootcourse.userservice.method1

************ IN METHOD2 OF USER DAO START**************

Propagation.REQUIRED_NEW: Is Parent transaction active: wait

Propagation.REQUIRED_NEW: Is Parent transaction name:
com.springbootcourse.userservice.userDao.method2

************ IN METHOD2 OF USER DAO END**************

IN METHOD1 of UserService Some initial DB OPERATION

IN METHOD1 of UserService Some Final DB OPERATION

3. SUPPORTS

@Transactional(propagation=Propagation.SUPPORTS)

IF [PARENT TXN PRESENT]

use it;

else

Execute the method without any transaction

```
 @Transactional(propagation= Propagation.SUPPORTS)
    public void method2(){
        boolean isTransactionActive =
TransactionSynchronizationManager.isActualTransactionActive();
        String currentTransactionName =
TransactionSynchronizationManager.getCurrentTransactionName();

        System.out.println("************ IN METHOD2 OF USER DAO START**************");

        System.out.println("Propagation.REQUIRED_NEW: Is Parent transaction active:
"+isTransactionActive );
        System.out.println("Propagation.REQUIRED_NEW: Is Parent transaction name:
"+currentTransactionName);


        System.out.println("************ IN METHOD2 OF USER DAO END**************");

    }
```

OUTPUT Hitting API:- http://localhost:8080/api/updateUser/Propagation_Required_Example

Inside Propagation_Required_Example() method of UserController.

IN METHOD1 of UserService IS Transaction active: true

IN METHOD1 IS Transaction name: com.springbootcourse.userservice.method1

************ IN METHOD2 OF USER DAO START**************

Propagation.REQUIRED_NEW: Is Parent transaction active: wait

Propagation.REQUIRED_NEW: Is Parent transaction name:
com.springbootcourse.userservice.userDao.method2

************ IN METHOD2 OF USER DAO END**************

IN METHOD1 of UserService Some initial DB OPERATION

IN METHOD1 of UserService Some Final DB OPERATION

4. NOT_SUPPORTS

@Transactional(propagation=Propagation.NOT_SUPPORTED)

IF [PARENT TXN PRESENT]

Suspend the parent txn;

Executed the method without any transactions;

Resume the parent txn;

else

Execute the method without any transaction

5.MANDATORY

@Transactional(propagation=Propagation.MANDATORY)

IF [PARENT TXN PRESENT]

USE IT

else

THROW EXCEPTION

5.NEVER

@Transactional(propagation=Propagation.NEVER)

IF [PARENT TXN PRESENT]

THROW EXCEPTION

else

Execute the method without any transaction

Declarative way of usage:

```java
@Component
public class UserDeclarative {

    @Autowired
    UserDAO userDAOobj;

    @Transactional
    public void updateUser() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());

        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagation();
        System.out.println("Some final DB operation");
    }

    public void updateUserFromNonTransactionalMethod() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagation();
        System.out.println("Some final DB operation");
    }
}
```

```java
@Component
public class UserDAO {

    @Transactional(propagation = Propagation.REQUIRED)
    /**
     * if(parent txn present)
     *        use it
     * else
     *      create new
     *
     */
    public void dbOperationWithRequiredPropagation() {

        //EXECUTE DB QUERIES
        boolean isTransactionActive = TransactionSynchronizationManager.isActualTransactionActive();
        String currentTransactionName = TransactionSynchronizationManager.getCurrentTransactionName();
        System.out.println("********************************");
        System.out.println("Propagation.REQUIRED: Is transaction active: " + isTransactionActive);
        System.out.println("Propagation.REQUIRED: Current transaction name: " + currentTransactionName);
        System.out.println("********************************");

    }
}
```

Output:

```
Is transaction active: true
Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
Some initial DB operation
********************************
Propagation.REQUIRED: Is parent transaction active: true
Propagation.REQUIRED: Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
********************************
Some final DB operation
```

```
Is transaction active: false
Current transaction name: null
Some initial DB operation
********************************
Propagation.REQUIRED: Is parent transaction active: true
Propagation.REQUIRED: Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDAO.dbOperationWithRequiredPropagation
********************************
Some final DB operation
```

Programmatic way of usage:

(Approach 1)

```java
@Component
public class UserDeclarative {

    @Autowired
    UserDAO userDAOobj;

    @Transactional
    public void updateUser() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());

        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagationUsingProgrammaticApproach1();
        System.out.println("Some final DB operation");
    }

    public void updateUserFromNonTransactionalMethod() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagationUsingProgrammaticApproach1();
        System.out.println("Some final DB operation");
    }
}
```

```java
@Component
public class UserDAO {

    PlatformTransactionManager userTransactionManager;

    UserDAO(PlatformTransactionManager userTransactionManager) {
        this.userTransactionManager = userTransactionManager;
    }

    /**
     * if(parent txn present)
     *        use it
     * else
     *      create new
     */
    public void dbOperationWithRequiredPropagationUsingProgrammaticApproach1(){
        DefaultTransactionDefinition transactionDefinition = new DefaultTransactionDefinition();
        transactionDefinition.setName("Testing REQUIRED propagation");
        transactionDefinition.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
        TransactionStatus status = userTransactionManager.getTransaction(transactionDefinition);
        try {
            //EXECUTE operation
            System.out.println("******************************");
            System.out.println("Propagation.REQUIRED: Is  transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
            System.out.println("Propagation.REQUIRED: Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
            System.out.println("******************************");
            userTransactionManager.commit(status);
        }
        catch (Exception e) {
            userTransactionManager.rollback(status);
        }
    }
```

(Approach 2) : using TransactionTemplate

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:testdb");
        dataSource.setUsername("sa2");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager userTransactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean
    public TransactionTemplate transactionTemplate(PlatformTransactionManager userTransactionManager) {
        TransactionTemplate transactionTemplate = new TransactionTemplate(userTransactionManager);
        transactionTemplate.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
        transactionTemplate.setName("TRANSACTION TEMPLATE REQUIRED PROPAGATION");
        return transactionTemplate;
    }
}
```

```java
@Component
public class UserService {

    @Autowired
    UserDAO userDAOobj;

    @Transactional
    public void updateUser() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());

        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagationUsingProgrammaticApproach2();
        System.out.println("Some final DB operation");
    }

    public void updateUserFromNonTransactionalMethod() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagationUsingProgrammaticApproach2();
        System.out.println("Some final DB operation");
    }
}
```

```java
@Component
public class UserDAO {

    TransactionTemplate transactionTemplate;

    UserDAO(TransactionTemplate transactionTemplate) {
        this.transactionTemplate = transactionTemplate;
    }

    /**
     * if(parent txn present)
     * use it
     * else
     * create new
     */
    public void dbOperationWithRequiredPropagationUsingProgrammaticApproach2() {

        TransactionCallback<TransactionStatus> operations = (TransactionStatus status) -> {
            //some operations for this method
            System.out.println("*******************************");
            System.out.println("Propagation.REQUIRED: Is  transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
            System.out.println("Propagation.REQUIRED: Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
            System.out.println("*******************************");
            return status;
        };

        TransactionStatus status = transactionTemplate.execute(operations);
    }
}
```

Output:

```
Is transaction active: true
Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
Some initial DB operation
*******************************
Propagation.REQUIRED: Is  transaction active: true
Propagation.REQUIRED: Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
*******************************
Some final DB operation
```

```
Is transaction active: false
Current transaction name: null
Some initial DB operation
*******************************
Propagation.REQUIRED: Is  transaction active: true
Propagation.REQUIRED: Current transaction name: TRANSACTION TEMPLATE REQUIRED PROPAGATION
*******************************
Some final DB operation
```

Spring boot @Transactional Annotation - Part3 | Isolation Level and its different types

Isolation Level

- It tells, how the changes made by one transactions are visible to other transactions running in parallel.

```
@Transactional(propagation = Propagation.REQUIRED,isolation = Isolation.READ_COMMITTED)
public void updateUsername(){
    //some operation here
}
```

Isolation Level	Dirty Read Possible	Non-Repeatable Read Possible	Phantom Read Possible	
READ_UNCOMMITED	Yes [NOT SOVED]	Yes [NOT SOVED]	Yes [NOT SOVED]	Concurrency High
READ_COMMITED	NO [Solved]	Yes [NOT SOVED]	Yes [NOT SOVED]	^
REPEATABLE_COMMITED	NO [SOVED]	NO [SOVED]	Yes [NOT SOVED]	^
SERIALIZABLE	NO[SOVED	NO [SOVED]	No [SOVED]	Concurrency low

Default isolation level, depends upon the DB which we are using. Like Most relational Databases uses **READ_COMMITED as default isolation**, but again it depends up on DB TO DB

ISOLATION LEVEL	Locking Strategy

READ_UNCOMMITTED	**Read** : No Lock acquired **Write**: No Lock acquired
Read COMMITTED	**Read**: Shared Lock acquired and released as soon as read is done. **Write**: Exclusive lock acquired and keep till the end of transaction
Repeatable Read	**Read**: Shared Lock applied and release only end of the transaction **Write**: Exclusive Lock acquired and released only at the end of the transaction.
Serializable	Same As Repeatable Read Locking Strategy + apply Range Lock and lock is release only at the end of transaction

Serializable

Talking About Phantom Read Problem.

DB

Id	Status
1	free
4	free

Transaction

- Shared lock releases at the end of the transaction
- but it also put **range lock**
- it means put lock on ranges like **[1 to 4]**
- **it like [1,2,3,4]**
- In case serializable less concurrency.
- DB Transaction Manager is abstract at application layer

time	Transaction T1	Transaction T2

t1	begin txn	
t2	read id>0 and id<5	
t3		**t2 wants to insert row with id=2 NOT POSSIBLE BECAUSE OF RANGE LOCK**

1. **Dirty Read Problem**

- Transaction A reads the un-committed data of other transaction.
- And If other transaction get ROLLED BACK, the un-committed data which was read by Transaction A is known as Dirty Read Problem.

Time	Transaction A	Transaction B	DB Status
T1	BEGIN_TRANSACTION	BEGIN_TRANSACTION	id:123 Status : free
T2		Update ROW id: 123 Status = booked	id:123 Status : booked [Not committed by Transaction B yet
T3	READ ROW id: 123 [Got status = booked]		id:123 Status : booked [Not committed by Transaction B yet

T4		ROLL BACK	id:123 Status : free (Un - commited changes of txn B got roll back

Transaction A suffer from Dirty read problem

2. **Non - Repeatable Read Problem**

- If suppose Transaction A, reads the same row several times and there is a chance that it get different value, then it's known as Non-Repeatable Read Problem.

time	Transaction A	DB	
T1	BEGIN_TRANSACTION	ID: 1 STATUS: free	
T2	READ ROW ID: 1 READ STATUS : FREE	ID: 1 STATUS: free	
T3		ID: 1 STATUS: BOOKED	Some other Transaction changed and committed the changes
T4	READ ROW ID: 1 READ STATUS : BOOKED	ID: 1 STATUS: BOOKED	
T5	COMMIT		

3. **Phantom Read Problem**

- If suppose Transaction A, executes same query several times but there is a chance that rows returned are different. Than it's known as **Phantom Read problem.**

time	Transaction A	DB	
T1	BEGIN_TRANSACTION	ID: 1, Status Free ID: 3, Status Booked	
T2	READ ROW WHERE ID>0 AND ID<5 [READS 2 rows ID:1 AND ID:3	ID: 1, Status Free ID: 3, Status Booked	
T3		ID: 1, Status Free ID: 2, Status Free ID: 3, Status Booked	Some other Transaction inserted the row with ID: 2 and commited
T4	READ ROW WHERE ID>0 AND ID<5 [READS 3 rows ID:1, ID:2 AND ID:3	ID: 1, Status Free ID: 2, Status Free ID: 3, Status Booked	
T5	COMMIT		

DB Locking Types

- Locking make sure that, no other transaction update the locked rows.

SYMBOL	LOCK TYPE	Another Shared Lock	Another Exclusive Lock
S	HAVE Shared Lock	YES	NO
X	Have Exclusive Lock	NO	NO

DATA BASE

ID	Status
1	free
2	booked
3	free

Shared Lock (S) also known as READ LOCK.

- More than one Transaction can take read and apply shared lock Reading Purpose only.

- Let's T1 takes id: 1 for READ and put shared lock on it.
- Will T2 takes id: 1 for Read : - > **yes** it will and also put shared lock on it But only in case of reading.

Transaction T1	Transaction T2
(S) Read ID : 1	(s) Read Id : 1

Exclusive Lock (X) also known as READ LOCK.

- Once Exclusive lock is taken by a transaction, no other transaction will allow to write even

read also.

- Let's Exclusive lock by taken By transaction T1.
- Now T2 wants to read and write On Id: 1 will it able to do NO.

Transaction T1	Transaction T2
(X) ID : 1	

12. Spring Boot Transactional Part1:
https://notebook.zohopublic.in/public/notes/dcr5za43213ff940049e4857a26389eb54486
13. Spring Boot Transactional Part2:
https://notebook.zohopublic.in/public/notes/dcr5z827d4c4d67cf491494f30a7733eeb3e1
14. Spring Boot Transactional Part3:
https://notebook.zohopublic.in/public/notes/dcr5z613a71d83b66404e8692cfca4790232e