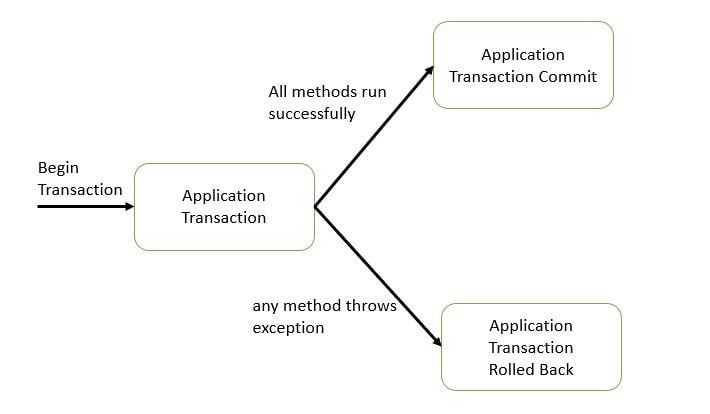
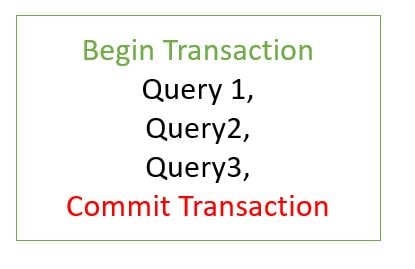
**Spring Boot Transaction Management**



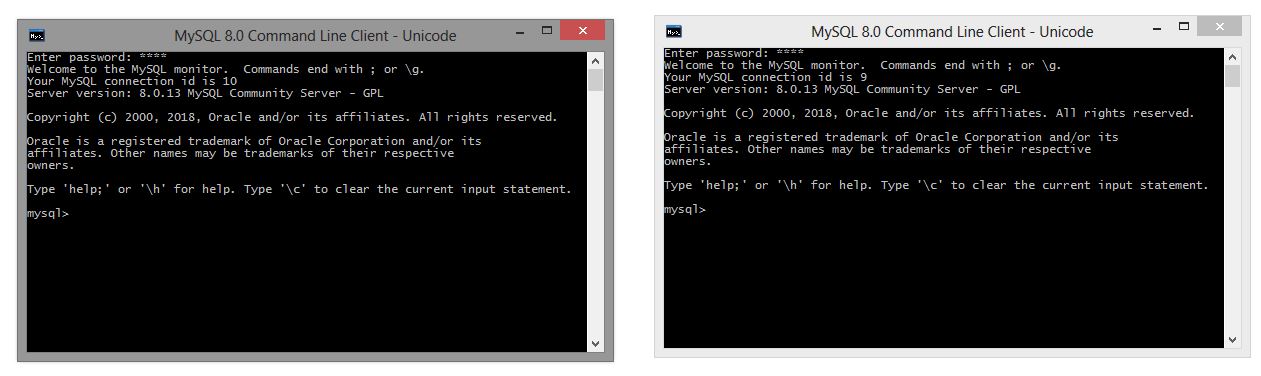
# Spring Boot Declarative Transaction Management Example

In previous tutorial - Spring Boot + JDBC Example we implemented JDBC using Spring boot with MySql database. In this tutorial we will be understanding what is transaction management and implement them for an application developed using Spring Boot + JDBC. In the next tutorial we will be implementing Transaction Propagation using Spring Boot.

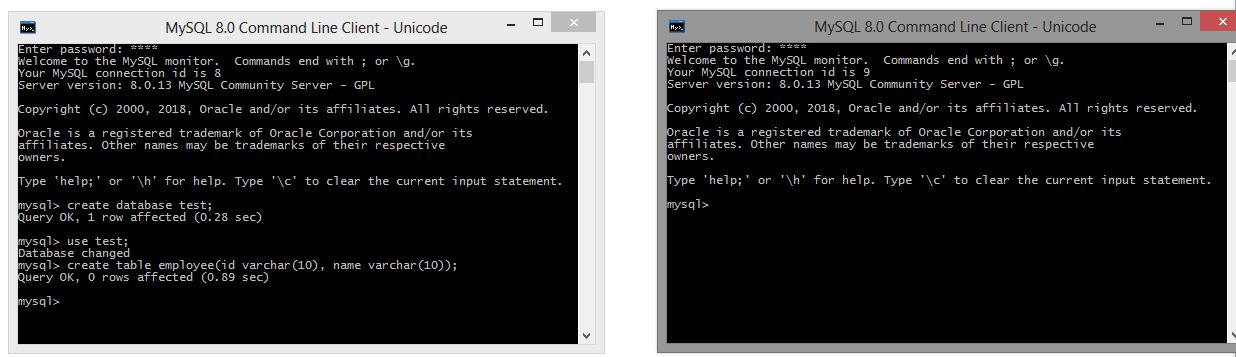
Lets Begin-

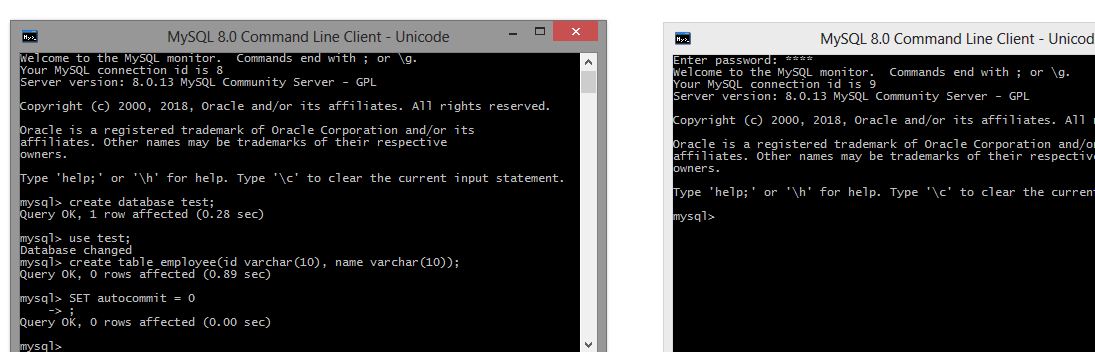
What are Database Transactions?  
A Database transaction is a single logical unit of work which accesses and possibly modifies the contents of a database.  
  


Let us check this for the mysql database.

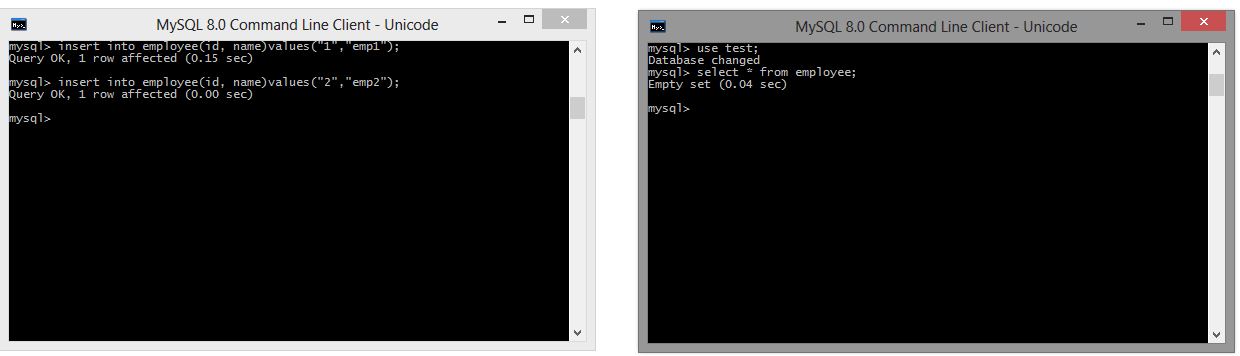
Open two separate windows for the mysql database.  


In one mysql window create a database named test and in it a table named employee

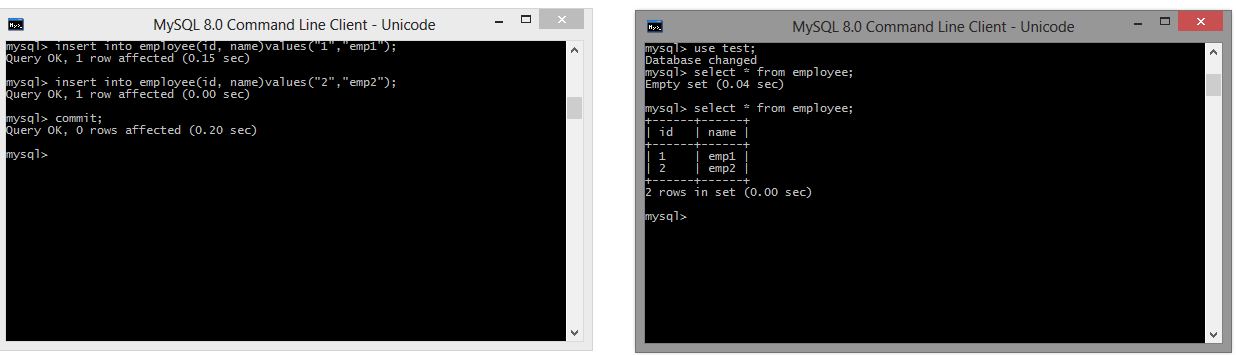


By default, the transactions are autocommit for mysql database.  
We will disable autocommit using following command- SET autocommit = 0  


In first mysql window use the following insert commands- If we now using the second mysql window do a select for the employee table we will not see any records. This is because the transactions are still not committed in the first mysql window.



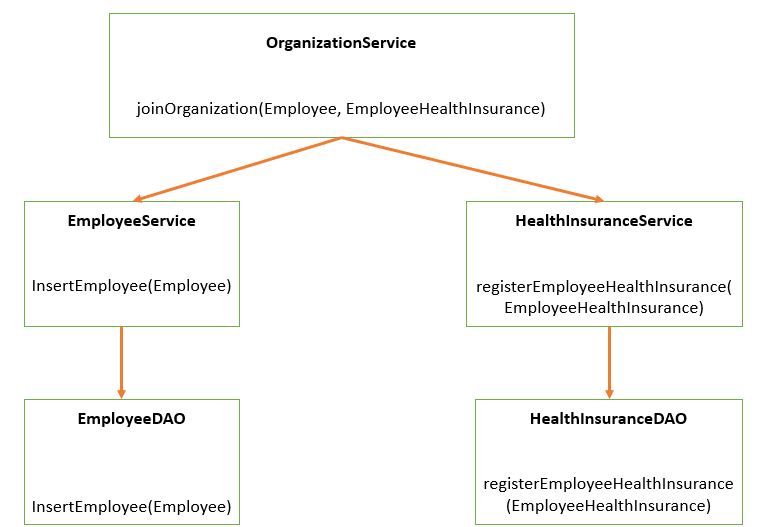
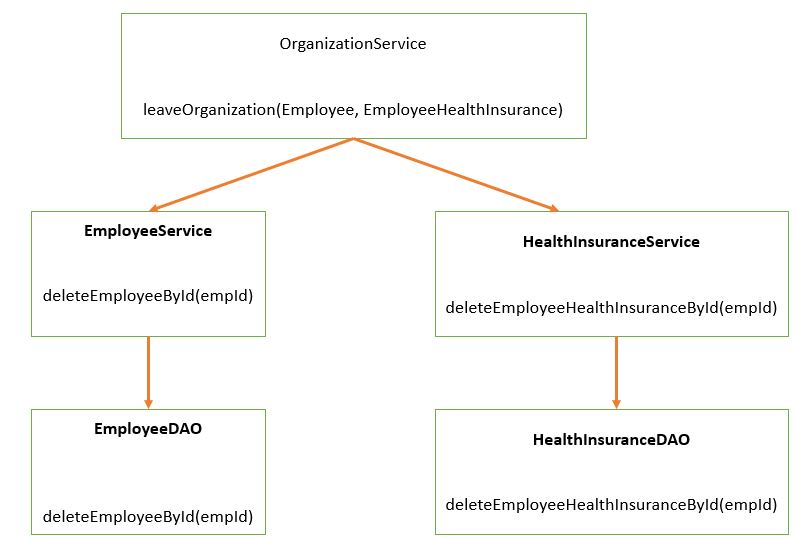
We now use the commit command in the first MySQL command. If we now using the second mysql window do a select for the employee table we will see the two records.



Let us now use application transaction for Spring Boot JDBC project.  
We will be developing a Spring Boot + JDBC project for employee management. It will be having 3 services-

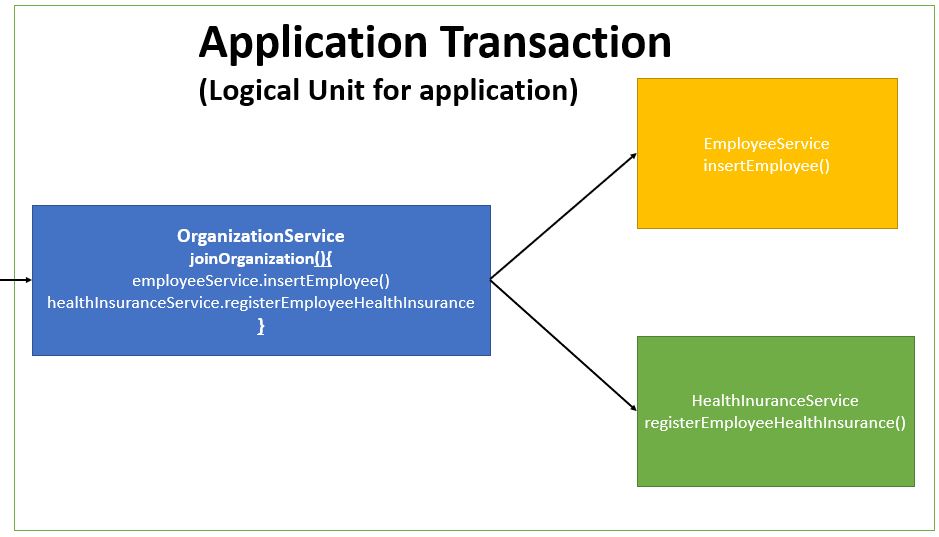
* **EmployeeService -** The service will perform Employee Operations
* **HealthInsuranceService -**The service will perform Employee Health Insurance Operations
* **OrganizationService -** The service will perform Organization Level Operation like Employee joining and exit. It makes use of the EmployeeService and HealthInsuranceService

## **OrganizationService Employee Join Workflow OrganizationService Employee Exit Workflow**

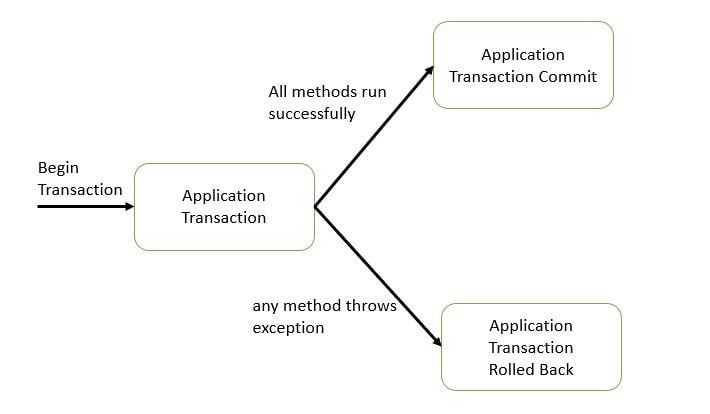
An application transaction is a sequence of application actions that are considered as a single logical unit by the application. For our application the joinOrganization method will be considered as one complete transaction. joinOrganization consists of two actions-

* Persist Employee Information
* Persist HealthInsurance Information

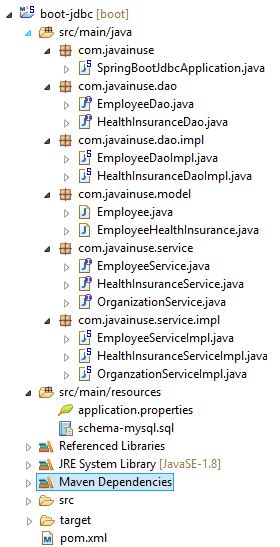


If due to any reason any one of the above action fails then the other action should also be roll backed.

So if Employee Information gets inserted but suppose due to some reason persist HealthInsurance is not successful, then Employee Information should also be rollbacked. It means it is all or none for a logical unit of work. Similar will be the case for exitOrganization Method which will be considered as one unit of work.



Initially we will not be using any transaction management. By default the spring boot transaction is auto commit. But this is not a good practice we will see why in the next section.

The maven project will be as follows-  


The pom.xml will be as follows-

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javainuse</groupId>

<artifactId>boot-jdbc</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>boot-jdbc</name>

<description>Demo project for Spring Boot</description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.1.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-jdbc</artifactId>**

**</dependency>**

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Create the application.properties as follows –

spring.datasource.url=jdbc:mysql://localhost/bootdb?createDatabaseIfNotExist=true&autoReconnect=true&useSSL=false

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.platform=mysql

spring.datasource.initialization-mode=always

logging.level.org.springframework=DEBUG

Create the schema-mysql.sql as follows. This is the initialization script which is run at the beginning by Spring Boot JDBC-

DROP TABLE IF EXISTS employee;

DROP TABLE IF EXISTS employeeHealthInsurance;

CREATE TABLE employee (

empId VARCHAR(10) NOT NULL,

empName VARCHAR(100) NOT NULL

);

CREATE TABLE employeeHealthInsurance (

empId VARCHAR(10) NOT NULL,

healthInsuranceSchemeName VARCHAR(100) NOT NULL,

coverageAmount VARCHAR(100) NOT NULL

);

Define the Model class Employee which will represent the Employee details-

package com.javainuse.model;

public class Employee {

private String empId;

private String empName;

public String getEmpId() {

return empId;

}

public void setEmpId(String empId) {

this.empId = empId;

}

public String getEmpName() {

return empName;

}

public void setEmpName(String empName) {

this.empName = empName;

}

@Override

public String toString() {

return "Employee [empId=" + empId + ", empName=" + empName + "]";

}

}

Create the EmployeeDAO interface for performing Employee operations as follows-

package com.javainuse.dao;

import com.javainuse.model.Employee;

public interface EmployeeDao {

void insertEmployee(Employee cus);

void deleteEmployeeById(String empid);

}

Create the EmployeeDAOImpl which implements the EmployeeDAO interface as follows. Spring Boot will detect spring-jdbc on the classpath and mysql and will create a DataSource and a JdbcTemplate for us automatically. Because such infrastructure is now available and we have no dedicated configuration, a DataSourceTransactionManager will also be created for us.

package com.javainuse.dao.impl;

import javax.annotation.PostConstruct;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.support.JdbcDaoSupport;

import org.springframework.stereotype.Repository;

import com.javainuse.dao.EmployeeDao;

import com.javainuse.model.Employee;

@Repository

public class EmployeeDaoImpl extends JdbcDaoSupport implements EmployeeDao {

@Autowired

DataSource dataSource;

@PostConstruct

private void initialize() {

setDataSource(dataSource);

}

@Override

public void insertEmployee(Employee emp) {

String sql = "INSERT INTO employee " + "(empId, empName) VALUES (?, ?)";

getJdbcTemplate().update(sql, new Object[] { emp.getEmpId(), emp.getEmpName() });

}

@Override

public void deleteEmployeeById(String empid) {

String sql = "DELETE FROM employee WHERE empId = ?";

getJdbcTemplate().update(sql, new Object[] { empid });

}

}

Define the Model class EmployeeHealthInsurance which will represent the Employee Health Insurance details-

package com.javainuse.model;

public class EmployeeHealthInsurance {

private String empId;

private String healthInsuranceSchemeName;

private int coverageAmount;

public String getEmpId() {

return empId;

}

public void setEmpId(String empId) {

this.empId = empId;

}

public String getHealthInsuranceSchemeName() {

return healthInsuranceSchemeName;

}

public void setHealthInsuranceSchemeName(String healthInsuranceSchemeName) {

this.healthInsuranceSchemeName = healthInsuranceSchemeName;

}

public int getCoverageAmount() {

return coverageAmount;

}

public void setCoverageAmount(int coverageAmount) {

this.coverageAmount = coverageAmount;

}

}

Create the HealthInsuranceDao for performing health insurance operations as follows-

package com.javainuse.dao;

import com.javainuse.model.EmployeeHealthInsurance;

public interface HealthInsuranceDao {

void registerEmployeeHealthInsurance(EmployeeHealthInsurance employeeHealthInsurance);

void deleteEmployeeHealthInsuranceById(String empid);

}

Create the EmployeeHealthInsuranceDAOImpl which implements HealthInsuranceDao as follows-

package com.javainuse.dao.impl;

import javax.annotation.PostConstruct;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.support.JdbcDaoSupport;

import org.springframework.stereotype.Repository;

import com.javainuse.dao.HealthInsuranceDao;

import com.javainuse.model.EmployeeHealthInsurance;

@Repository

public class HealthInsuranceDaoImpl extends JdbcDaoSupport implements HealthInsuranceDao {

@Autowired

DataSource dataSource;

@PostConstruct

private void initialize() {

setDataSource(dataSource);

}

@Override

public void registerEmployeeHealthInsurance(EmployeeHealthInsurance emp) {

String sql = "INSERT INTO employeeHealthInsurance "

+ "(empId, healthInsuranceSchemeName, coverageAmount) VALUES (?, ?,?)";

getJdbcTemplate().update(sql,

new Object[] { emp.getEmpId(), emp.getHealthInsuranceSchemeName(), emp.getCoverageAmount() });

}

@Override

public void deleteEmployeeHealthInsuranceById(String empid) {

String sql = "DELETE FROM employeeHealthInsurance WHERE empId = ?";

getJdbcTemplate().update(sql, new Object[] { empid });

}

}

Create the EmployeeService interface for performing employee operations as follows-

package com.javainuse.service;

import com.javainuse.model.Employee;

public interface EmployeeService {

void insertEmployee(Employee emp);

void deleteEmployeeById(String empid);

}

Create the EmployeeServiceImpl which implements the EmployeeService as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.javainuse.dao.EmployeeDao;

import com.javainuse.model.Employee;

import com.javainuse.service.EmployeeService;

@Service

public class EmployeeServiceImpl implements EmployeeService {

@Autowired

EmployeeDao employeeDao;

@Override

public void insertEmployee(Employee employee) {

employeeDao.insertEmployee(employee);

}

@Override

public void deleteEmployeeById(String empid) {

employeeDao.deleteEmployeeById(empid);

}

}

Create the HealthInsuranceService interface as follows-

package com.javainuse.service;

import com.javainuse.model.EmployeeHealthInsurance;

public interface HealthInsuranceService {

void registerEmployeeHealthInsurance(EmployeeHealthInsurance employeeHealthInsurance);

void deleteEmployeeHealthInsuranceById(String empid);

}

Create the HealthInsuranceServiceImpl which implements the HealthInsuranceService as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.javainuse.dao.HealthInsuranceDao;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.HealthInsuranceService;

@Service

public class HealthInsuranceServiceImpl implements HealthInsuranceService {

@Autowired

HealthInsuranceDao healthInsuranceDao;

@Override

public void registerEmployeeHealthInsurance(EmployeeHealthInsurance employeeHealthInsurance) {

healthInsuranceDao.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

public void deleteEmployeeHealthInsuranceById(String empid) {

healthInsuranceDao.deleteEmployeeHealthInsuranceById(empid);

}

}

Create the OrganizationService interface as follows-

package com.javainuse.service;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

public interface OrganizationService {

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance);

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance);

}

Create the OrganizationServiceImpl which implements the OrganizationService. It makes use of the EmployeeService and the HealthInsuranceService.

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.insertEmployee(employee);

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}

Finally create the Spring Boot Main class as follows-

package com.javainuse;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.OrganizationService;

@SpringBootApplication

public class SpringBootJdbcApplication {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(SpringBootJdbcApplication.class, args);

OrganizationService organizationService = context.getBean(OrganizationService.class);

Employee emp = new Employee();

emp.setEmpId("emp1");

emp.setEmpName("emp1");

EmployeeHealthInsurance employeeHealthInsurance = new EmployeeHealthInsurance();

employeeHealthInsurance.setEmpId("emp1");

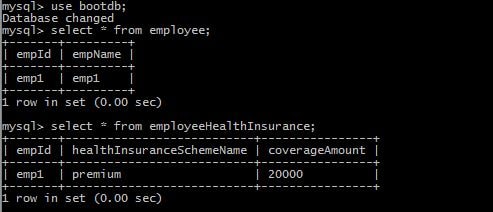
employeeHealthInsurance.setHealthInsuranceSchemeName("premium");

employeeHealthInsurance.setCoverageAmount(20000);

organizationService.joinOrganization(emp, employeeHealthInsurance);

}

}

If we now run the application, record will be inserted in both the employee table and the employeehealthinsurance table  
  
Suppose the employeeService call is successful but due to some reason the healthInsuranceService call fails. What should happen in this case. In such a scenario the entry made in the employee table for the new employee should also be reverted. Let us see how our application will behave in such a scenario.  
We are manually throwing an unchecked exception after the first service call is made.

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.insertEmployee(employee);

**if (employee.getEmpId().equals("emp1")) {**

**throw new RuntimeException("thowing exception to test transaction rollback");**

**}**

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

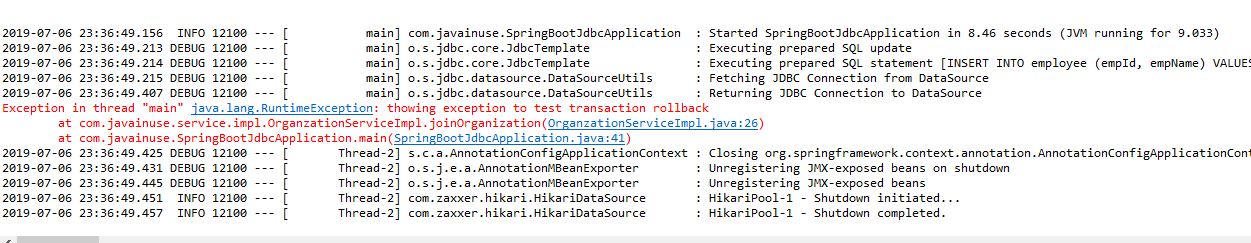
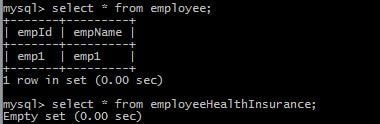
public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}

Lets now run the application.  
  
We can see that there is record in employee table but not in employeehealthinsurance table.  
  
Now let us implement transaction management. We will be using the Transactional annotation. Transaction is a cross cutting concern and it is implemented using AOP in Spring Boot.  


package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

**@Transactional**

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.insertEmployee(employee);

if (employee.getEmpId().equals("emp1")) {

throw new RuntimeException("thowing exception to test transaction rollback");

}

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

**@Transactional**

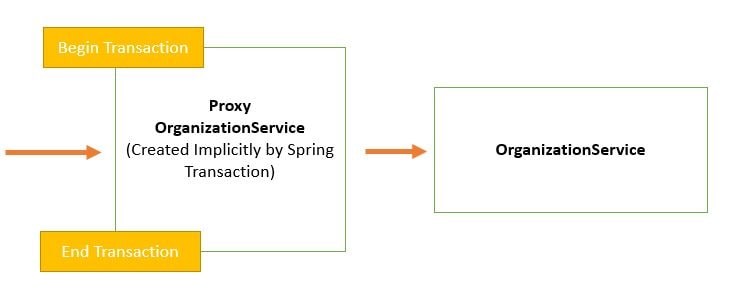
public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

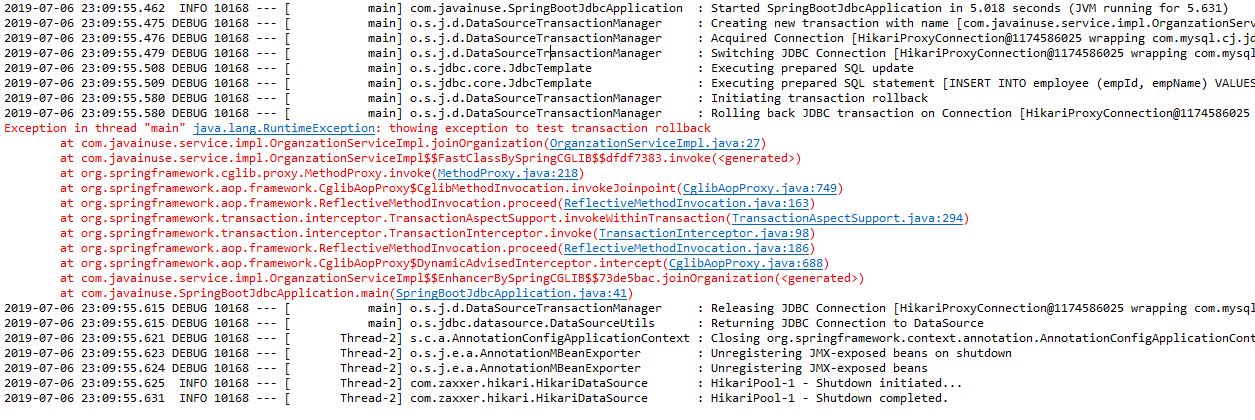
healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

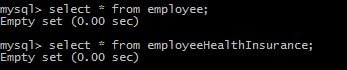
}

Spring Boot implicitly creates a proxy for the transaction annotated methods. So for such methods the proxy acts like a wrapper which takes care of creating a transaction at the beginning of the method call and committing the transaction after the method is executed.  


The component that intercepts the @Transactional annotated method like the EmployeeService. Now let us run the application again.



If we now check the employee and the employeehealthinsurance table there are no records in both so our records are getting roll backed correctly.

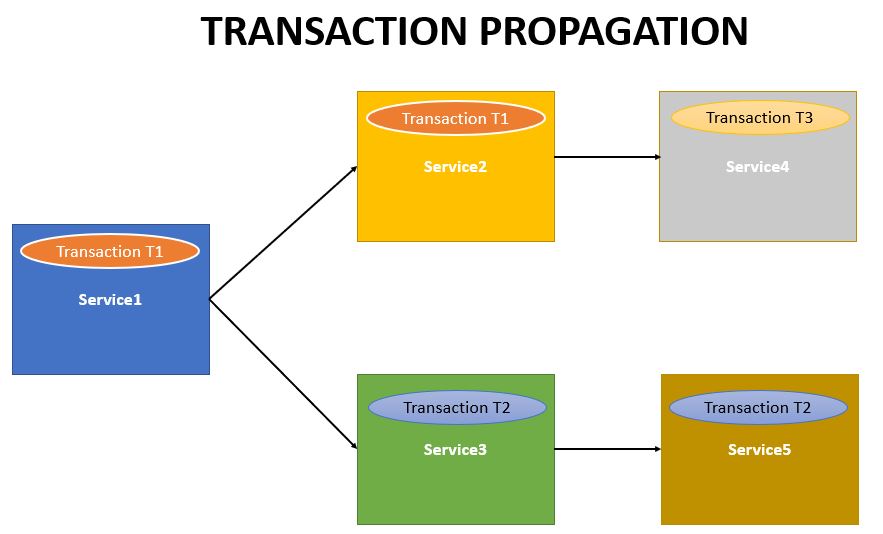


# Spring Boot Transactions - Understanding Transaction Propagation

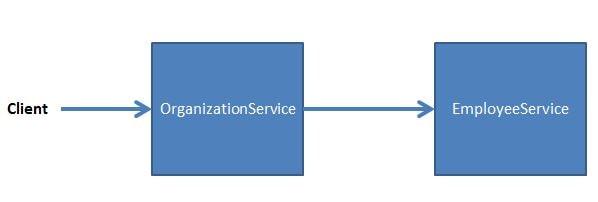
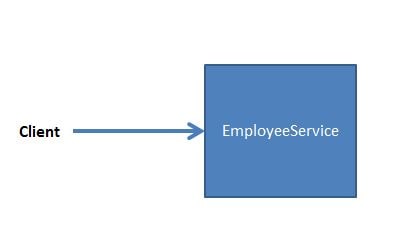
In previous tutorial - Spring Boot Transaction Management Example we saw what are transactions and implemented declarative transaction management. In this tutorial we will be understanding what is propagation and its different types.

Lets Begin-

What is Transaction Propagation?  
Any application involves a number of services or components making a call to other services or components. Transaction Propagation indicates if any component or service will or will not participate in transaction and how will it behave if the callingcomponent/service already has or does not have a transaction created already.



We will be making use of the Spring Boot Transaction Project we developed in previous chapter. It had the Organization service which makes a call to the Employee Service and the Health Insurance Service.  
  
But suppose the user wants to call the Employee Service in both ways i.e.

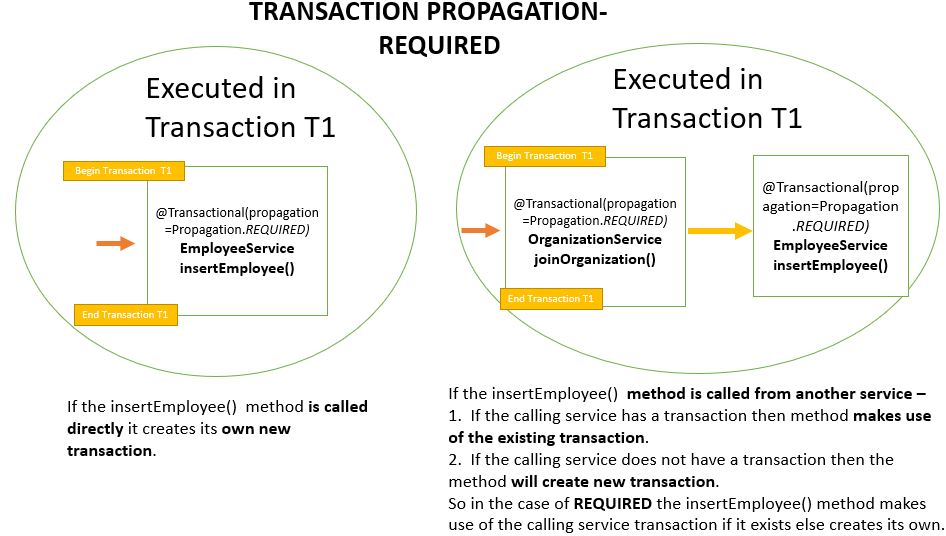
* Call using Organization service  
  
* Call the Employee Service directly.  
  

As the Employee Service may also be called directly, we will need to use Transaction annotation with Employee Service also. So, both the services - Organization Service and the Employee Service will be using Transaction annotation.

We will be looking at the various propagation scenarios by observing the behaviour of the Organization and Employee service. There are six types of Transaction Propagations-

* **REQUIRED**
* **SUPPORTS**
* **NOT\_SUPPORTED**
* **REQUIRES\_NEW**
* **NEVER**
* **MANDATORY**

## **Transaction Propagation -**REQUIRED**(Default Transaction Propagation)**



Here both the Organization Service and the Employee Service have the transaction propagation defined as Required. This is the default Transaction Propagation.  
Code-

The Organization Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

**@Transactional**

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.insertEmployee(employee);

if (employee.getEmpId().equals("emp1")) {

throw new RuntimeException("thowing exception to test transaction rollback");

}

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}

The Employee Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.dao.EmployeeDao;

import com.javainuse.model.Employee;

import com.javainuse.service.EmployeeService;

@Service

**@Transactional**

public class EmployeeServiceImpl implements EmployeeService {

@Autowired

EmployeeDao employeeDao;

@Override

public void insertEmployee(Employee employee) {

employeeDao.insertEmployee(employee);

}

@Override

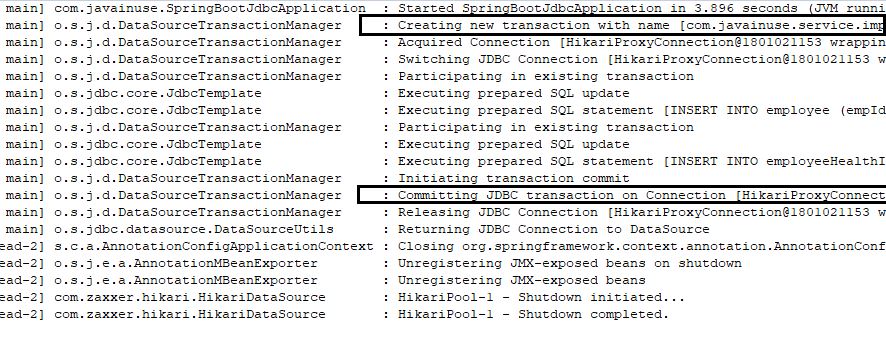
public void deleteEmployeeById(String empid) {

employeeDao.deleteEmployeeById(empid);

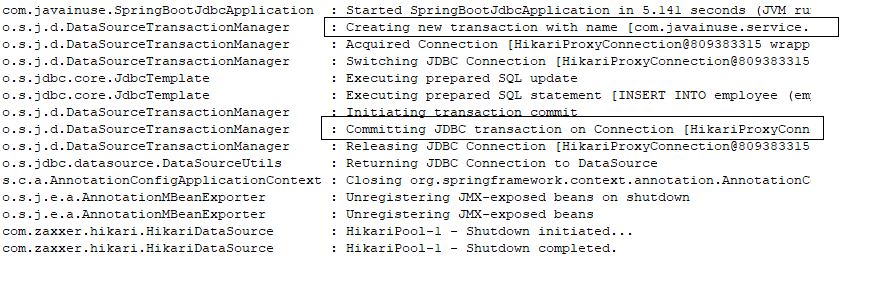
}

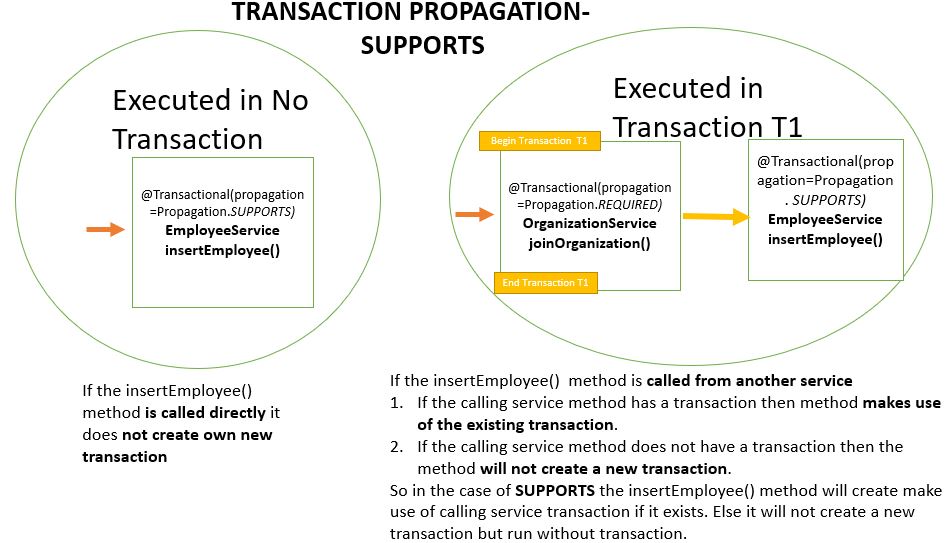
}

Output  
EmployeeService called using OrganizationService –



EmployeeService called directly -

  
**Transaction Propagation -**SUPPORTS



Here both the Organization Service has the transaction propagation defined as Required while Employee Service the transaction propagation is defined as Supports.  
Code-  
The Organization Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

**@Transactional**

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.insertEmployee(employee);

if (employee.getEmpId().equals("emp1")) {

throw new RuntimeException("thowing exception to test transaction rollback");

}

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}

The Employee Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.dao.EmployeeDao;

import com.javainuse.model.Employee;

import com.javainuse.service.EmployeeService;

@Service

**@Transactional(propagation=Propagation.SUPPORTS)**

public class EmployeeServiceImpl implements EmployeeService {

@Autowired

EmployeeDao employeeDao;

@Override

public void insertEmployee(Employee employee) {

employeeDao.insertEmployee(employee);

}

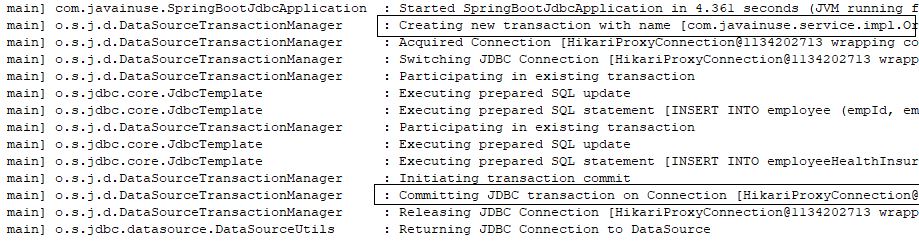
@Override

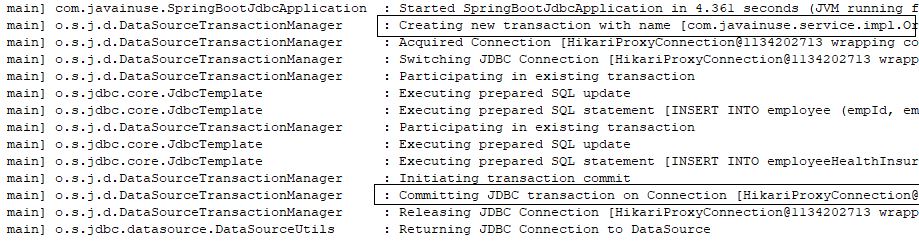
public void deleteEmployeeById(String empid) {

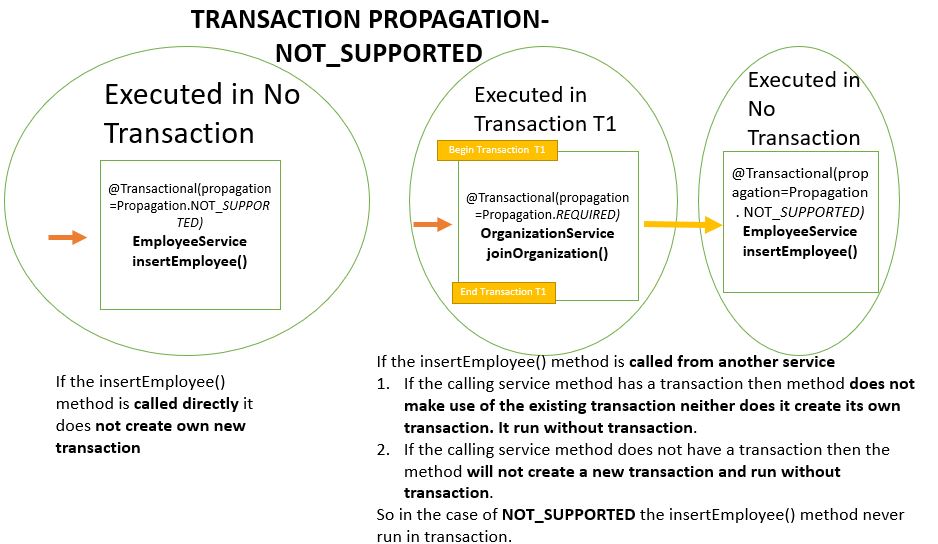
employeeDao.deleteEmployeeById(empid);

}

}

Output  
EmployeeService called using OrganizationService **-** 

**EmployeeService called directly -**  
**Transaction Propagation -**NOT\_SUPPORTED



Here for the Organization Service we have defined the transaction propagation as REQUIRED and the Employee Service have the transaction propagation defined as **NOT\_SUPPORTED**

Code-  
The Organization Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

**@Transactional**

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.insertEmployee(employee);

if (employee.getEmpId().equals("emp1")) {

throw new RuntimeException("thowing exception to test transaction rollback");

}

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}

The Employee Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.dao.EmployeeDao;

import com.javainuse.model.Employee;

import com.javainuse.service.EmployeeService;

@Service

**@Transactional(propagation=Propagation.NOT\_SUPPORTED)**

public class EmployeeServiceImpl implements EmployeeService {

@Autowired

EmployeeDao employeeDao;

@Override

public void insertEmployee(Employee employee) {

employeeDao.insertEmployee(employee);

}

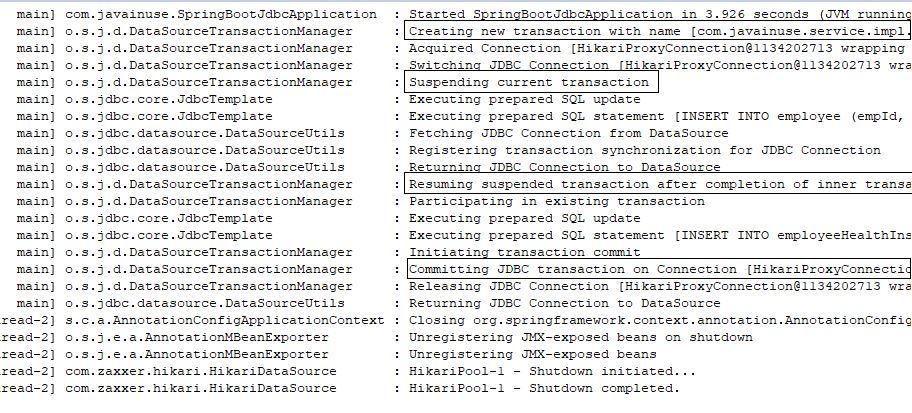
@Override

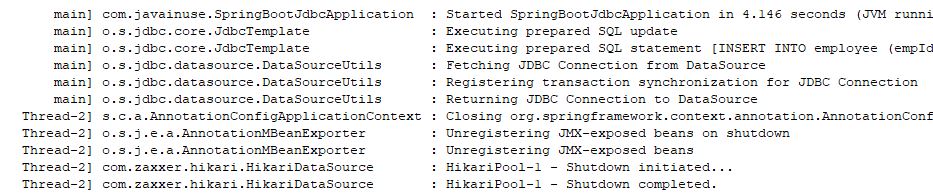
public void deleteEmployeeById(String empid) {

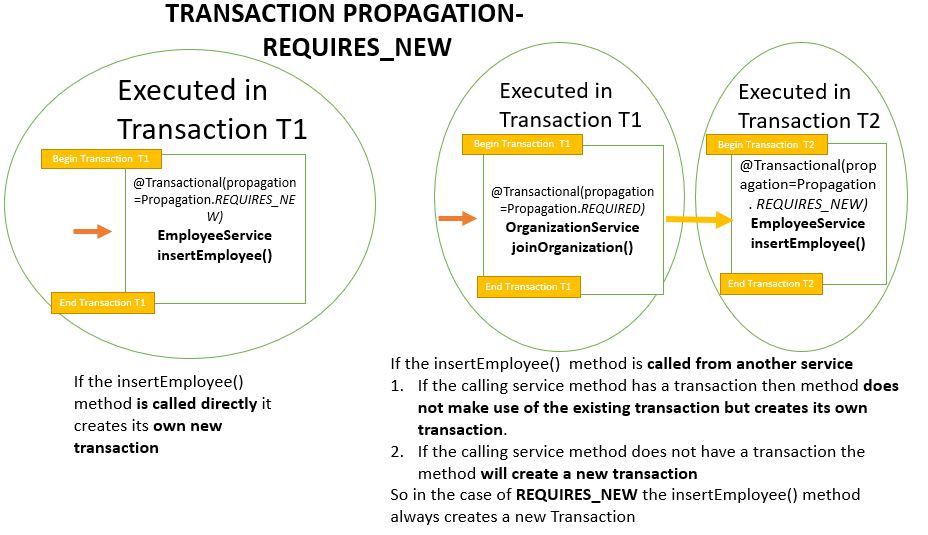
employeeDao.deleteEmployeeById(empid);

}

}

**Output  
EmployeeService called using OrganizationService -**

**EmployeeService called directly –**  
  
**Transaction Propagation -**REQUIRES\_NEW



Here for the Organization Service we have defined the transaction propagation as REQUIRED and the Employee Service have the transaction propagation defined as REQUIRES\_NEW  
Code-  
The Organization Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

**@Transactional**

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.insertEmployee(employee);

if (employee.getEmpId().equals("emp1")) {

throw new RuntimeException("thowing exception to test transaction rollback");

}

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}

The Employee Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.dao.EmployeeDao;

import com.javainuse.model.Employee;

import com.javainuse.service.EmployeeService;

@Service

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

public class EmployeeServiceImpl implements EmployeeService {

@Autowired

EmployeeDao employeeDao;

@Override

public void insertEmployee(Employee employee) {

employeeDao.insertEmployee(employee);

}

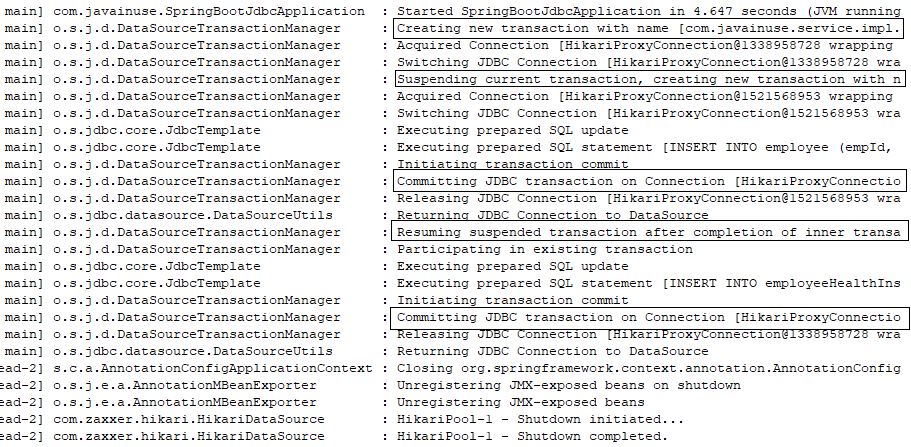
@Override

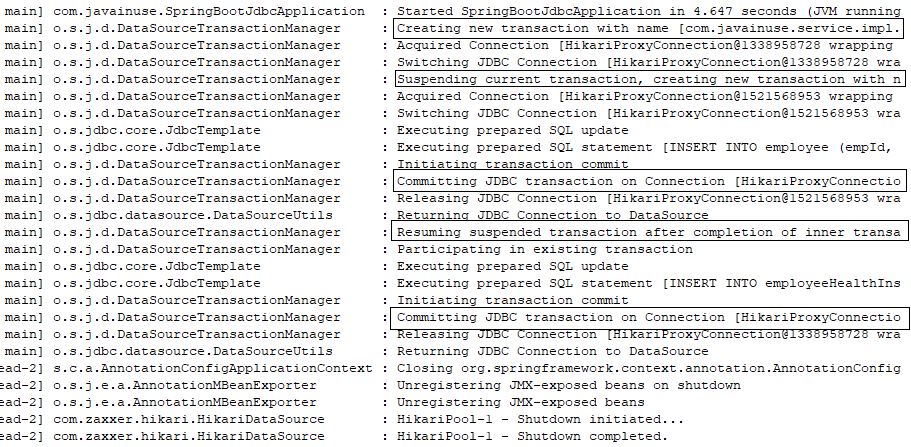
public void deleteEmployeeById(String empid) {

employeeDao.deleteEmployeeById(empid);

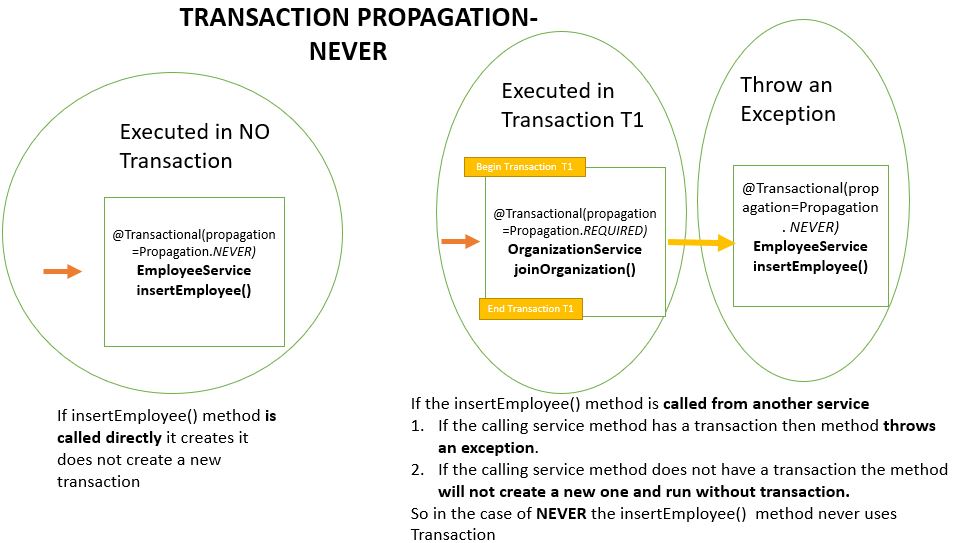
}

}

Output  
EmployeeService called using OrganizationService -  


EmployeeService called directly -  


## **Transaction Propagation -**NEVER



Here for the Organization Service, we have defined the transaction propagation as REQUIRED and the Employee Service have the transaction propagation defined as NEVERs

Code-  
The Organization Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

**@Transactional**

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.insertEmployee(employee);

if (employee.getEmpId().equals("emp1")) {

throw new RuntimeException("thowing exception to test transaction rollback");

}

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}

The Employee Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.dao.EmployeeDao;

import com.javainuse.model.Employee;

import com.javainuse.service.EmployeeService;

@Service

**@Transactional(propagation=Propagation.NEVER)**

public class EmployeeServiceImpl implements EmployeeService {

@Autowired

EmployeeDao employeeDao;

@Override

public void insertEmployee(Employee employee) {

employeeDao.insertEmployee(employee);

}

@Override

public void deleteEmployeeById(String empid) {

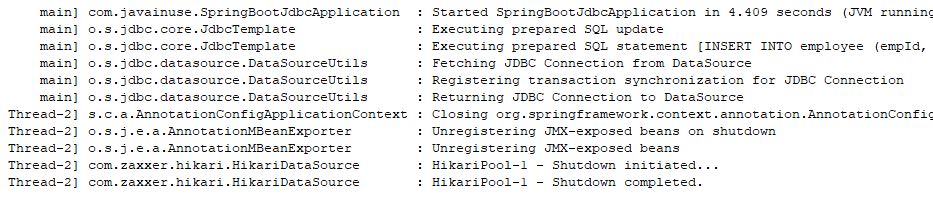
employeeDao.deleteEmployeeById(empid);

}

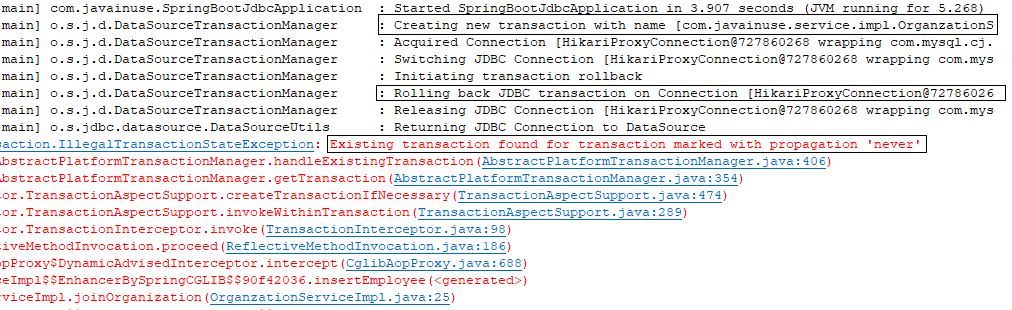
}

Output

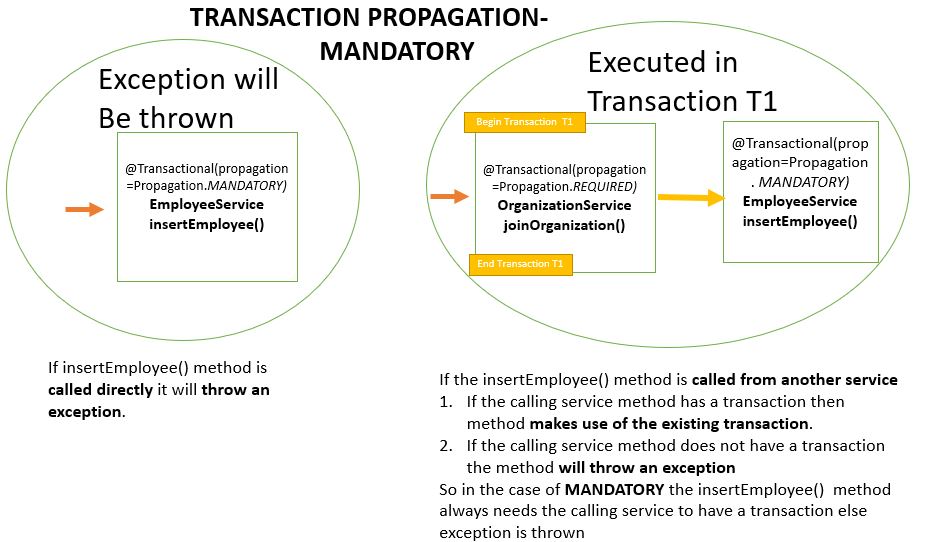
EmployeeService called using OrganizationService –



**EmployeeService called directly –**



## **Transaction Propagation -**MANDATORY



Here for the Organization Service, we have defined the transaction propagation as REQUIRED and the Employee Service have the transaction propagation defined as MANDATORY

Code-  
The Organization Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

**@Transactional**

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.insertEmployee(employee);

if (employee.getEmpId().equals("emp1")) {

throw new RuntimeException("thowing exception to test transaction rollback");

}

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}

The Employee Service will be as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.dao.EmployeeDao;

import com.javainuse.model.Employee;

import com.javainuse.service.EmployeeService;

@Service

**@Transactional(propagation=Propagation.MANDATORY)**

public class EmployeeServiceImpl implements EmployeeService {

@Autowired

EmployeeDao employeeDao;

@Override

public void insertEmployee(Employee employee) {

employeeDao.insertEmployee(employee);

}

@Override

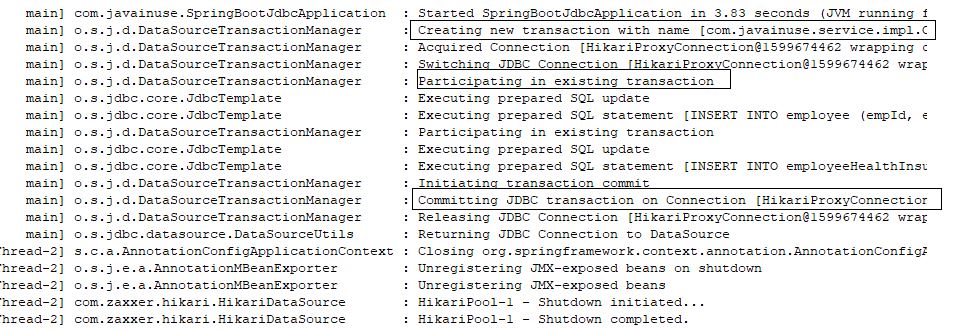
public void deleteEmployeeById(String empid) {

employeeDao.deleteEmployeeById(empid);

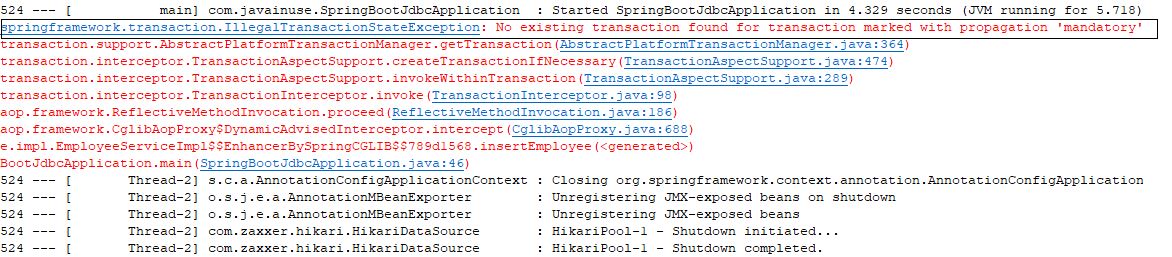
}

}

Output  
EmployeeService called using OrganizationService –



**EmployeeService** called directly –



So the summary will be as follows-

|  |  |
| --- | --- |
| **Propagation** | **Behaviour** |
| **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 |
| **SUPPORTS** | **It may or may not run in a transaction.** If current transaction exists then it is supported. If none exists then gets executed without transaction. |
| **NOT\_SUPPORTED** | **Always executes without a transaction.** If there is any existing transaction it gets suspended |
| **REQUIRES\_NEW** | **Always executes in a new transaction.** If there is any existing transaction it gets suspended |
| **NEVER** | **Always executes without any transaction.** It throws an exception if there is an existing transaction |
| **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. |

# Spring Boot Transaction Management - Transaction Rollback Example

# we saw what are transactions and implemented declarative transaction management. In a previous tutorial - [Spring Boot Transactions - Understanding Transaction Propagation](https://www.javainuse.com/spring/boot-transaction-propagation) we also looked at what is propagation and its different types using Spring Boot.

# In [Spring Boot Transaction Management Example](https://www.javainuse.com/spring/boot-transaction) we had seen application transaction is a sequence of application actions that are considered as a single logical unit by the application.

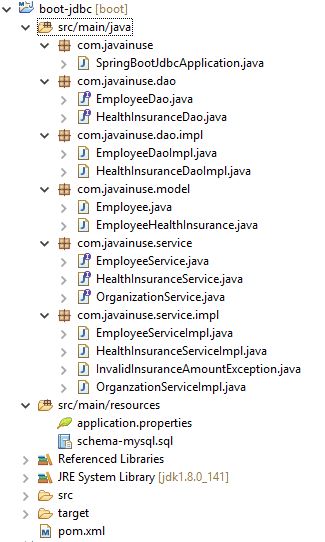
# OrganizationService Exit

# For an application transaction if any action fails then all other actions get rolled back. Previous [Transaction Management Example](https://www.javainuse.com/spring/boot-transaction) we had tested the rollback by throwing an unchecked exception.

# However in real time scenarios it is the checked exception that gets thrown. These are business exceptions based on some logic

# So how will our transactions behave in case of Checked Exceptions? In case of checked exceptions the previously executed transactions do not get rolled back automatically even if we have used transaction annotation. We need to inform the application how to handle roll back in event of checked exception. This is achieved using the Rollback For annotation

Lets Begin-

We will be modifying the code we had created previously for Transaction Management.  
We will create a custom checked exception called InvalidInsuranceAmountException. According to our business logic if the insurance coverage amount is less than zero then this exception should get thrown. The maven project will be as follows -  


Create the Custom Exception as follows-

package com.javainuse.service.impl;

public class InvalidInsuranceAmountException extends Exception {

private static final long serialVersionUID = 1L;

public InvalidInsuranceAmountException(String cause) {

super(cause);

}

}

The HealthInsuranceService interface throws this exception for the registerEmployeeHealthInsurance method.

package com.javainuse.service;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.impl.InvalidInsuranceAmountException;

public interface HealthInsuranceService {

void registerEmployeeHealthInsurance(EmployeeHealthInsurance employeeHealthInsurance)

**throws InvalidInsuranceAmountException**;

void deleteEmployeeHealthInsuranceById(String empid);

}

In the HealthInsuranceServiceImpl class for the registerEmployeeHealthInsurance we put a check for verifying if the coverage amount is less than zero. If it is then we throw the InvalidInsuranceAmountException.

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.dao.HealthInsuranceDao;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.HealthInsuranceService;

@Service

@Transactional

public class HealthInsuranceServiceImpl implements HealthInsuranceService {

@Autowired

HealthInsuranceDao healthInsuranceDao;

@Override

public void registerEmployeeHealthInsurance(EmployeeHealthInsurance employeeHealthInsurance)

throws InvalidInsuranceAmountException {

**if (employeeHealthInsurance.getCoverageAmount() < 0) {**

**throw new InvalidInsuranceAmountException("Coverage Amount Should not be negative");**

**}**

healthInsuranceDao.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

public void deleteEmployeeHealthInsuranceById(String empid) {

healthInsuranceDao.deleteEmployeeHealthInsuranceById(empid);

}

}

The OrganizationService interface for the joinOrganization method we throw the InvalidInsuranceAmountException.

package com.javainuse.service;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.impl.InvalidInsuranceAmountException;

public interface OrganizationService {

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance)

**throws InvalidInsuranceAmountException;**

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance);

}

In the OrganzationServiceImpl which makes a call to the HealthInsuranceService class, we catch the InvalidInsuranceAmountException log it and throw it again. This is because we want the calling client to know what exception has occurred.

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

@Transactional

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

@Transactional

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance)

**throws InvalidInsuranceAmountException** {

employeeService.insertEmployee(employee);

try {

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

} **catch (InvalidInsuranceAmountException e) {**

**throw new InvalidInsuranceAmountException("Exception is thrown");**

**}**

}

@Override

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}

Finally, in the SpringBootJdbcApplication we throw the InvalidInsuranceAmountException.

package com.javainuse;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.OrganizationService;

import com.javainuse.service.impl.InvalidInsuranceAmountException;

@SpringBootApplication

public class SpringBootJdbcApplication {

@Autowired

EmployeeService employeeService;

public static void main(String[] args) **throws InvalidInsuranceAmountException** {

ApplicationContext context = SpringApplication.run(SpringBootJdbcApplication.class, args);

OrganizationService organizationService = context.getBean(OrganizationService.class);

Employee emp= new Employee();

emp.setEmpId("emp1");

emp.setEmpName("emp1");

EmployeeHealthInsurance employeeHealthInsurance= new EmployeeHealthInsurance();

employeeHealthInsurance.setEmpId("emp1");

employeeHealthInsurance.setHealthInsuranceSchemeName("premium");

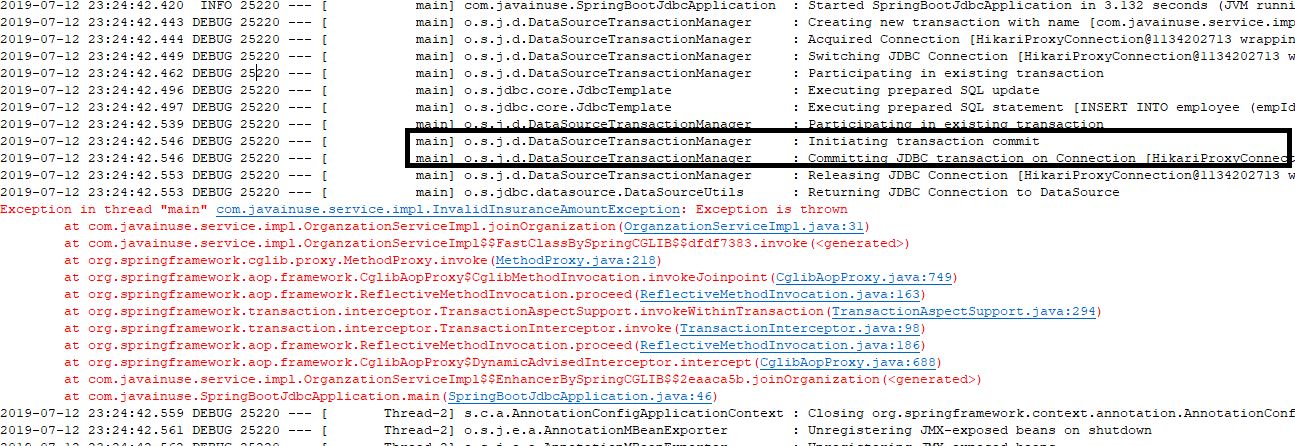
employeeHealthInsurance.setCoverageAmount(0);

organizationService.joinOrganization(emp, employeeHealthInsurance);

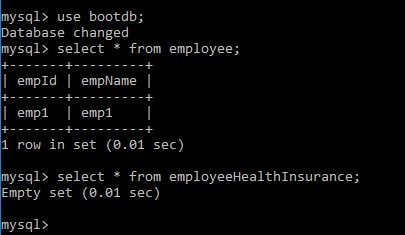
}

}

If now run the application- We see that the employeeService transaction is not rolled back due to an exception in employeeHealthService.



In the Database we see that the insert for employee table has not been rolledback-

  
But this should not be the case. To achieve roll back for checked exception we will need to specify it using Rollback for Annotation.

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

@Transactional

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

@Transactional(**rollbackFor = InvalidInsuranceAmountException.class**)

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance)

throws InvalidInsuranceAmountException {

employeeService.insertEmployee(employee);

try {

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

} catch (InvalidInsuranceAmountException e) {

throw new InvalidInsuranceAmountException("Exception is thrown");

}

}

@Override

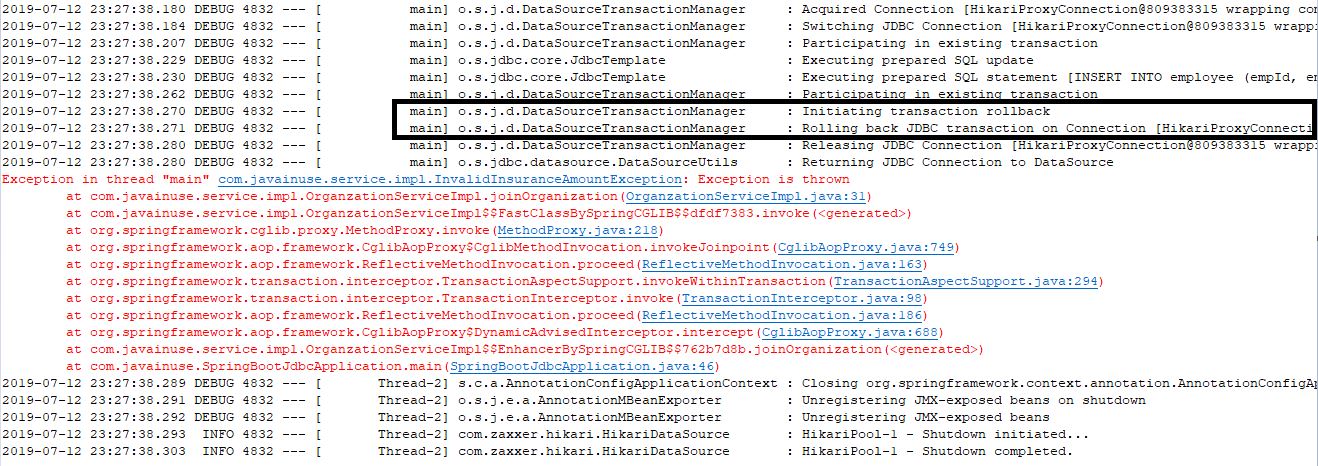
public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

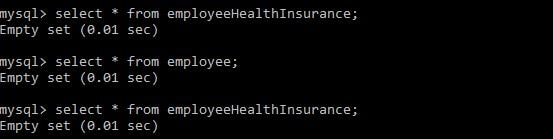
healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}

Now run the application again. We see that the employeeService transaction is rolled back due to an exception in employeeHealthService.  


In the Database we see that the insert for employee table has been rolledback-



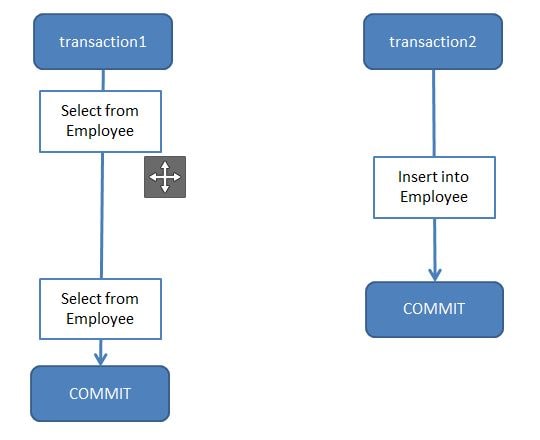
# Spring Boot - Transaction Isolation Tutorial

In previous tutorial - Spring Boot Transaction Management Example we saw what are transactions and implemented declarative transaction management. In this tutorial we will be understanding what is transaction isolation and its different types.

**Let’s Begin-**

**What is Transaction Isolation?**Transaction Isolation defines the database state when two transactions concurrently act on the same database entity. It involves locking of database records. So it describes the behaviour or state of the database when one transaction is working on database entity and then some other concurrent transaction tries to simultaneously access/edit the same database entity.

The ANSI/ISO standard defines four isolation levels. Isolation is one of the ACID (Atomicity, Consistency, Isolation, Durability) properties. So transaction isolation level is not something specific to Spring Framework. Using Spring we can change the isolation level to suit our business logic.



Before implementing Isolation Level using Spring, let us first understand isolation levels at Database level.

We will be creating a table name employee and using this table try understand the isolation levels-

CREATE TABLE employee (

empId VARCHAR(10) NOT NULL,

empName VARCHAR(100) NOT NULL

);

Some of the SQL commands I have used for implementing Isolation Levels are

//Show existing transaction isolation level if mysql version >= 8

**SELECT @@TRANSACTION\_ISOLATION;**

//Show existing transaction isolation level if mysql version < 8

**SELECT @@TX\_ISOLATION;**

//Set transaction isolation level to serializable. Using same syntax

//we can set it to other isolation level.

**SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;**

//By default auto commit is enabled for mysql transaction. So we will disable it.

**SET AUTOCOMMIT=0;**

//Start transaction

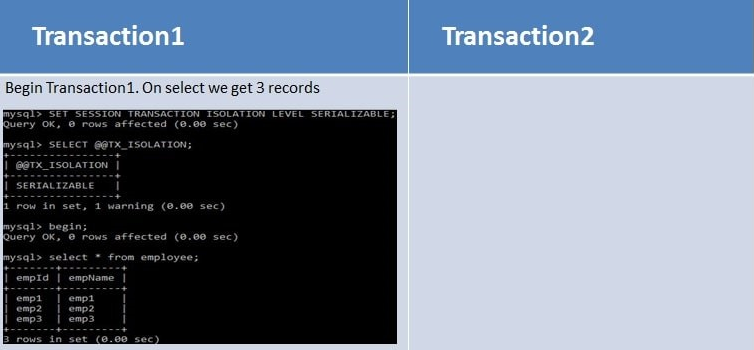
**BEGIN**

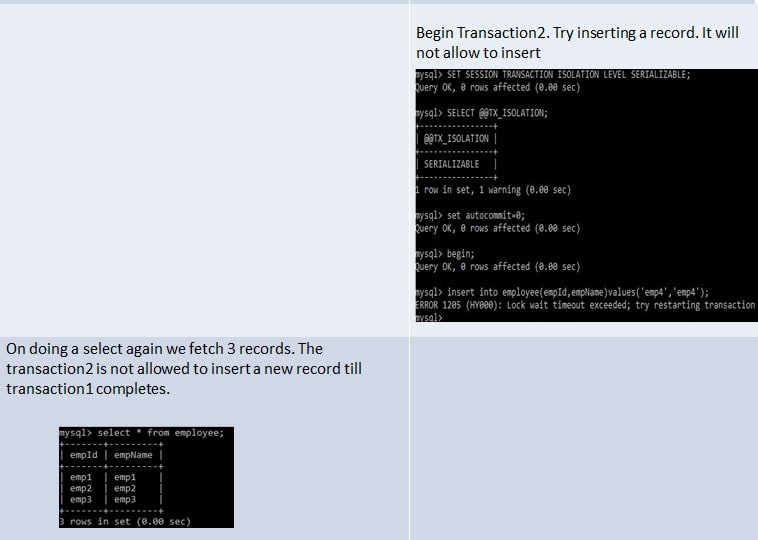
//Commit transaction

**COMMIT**

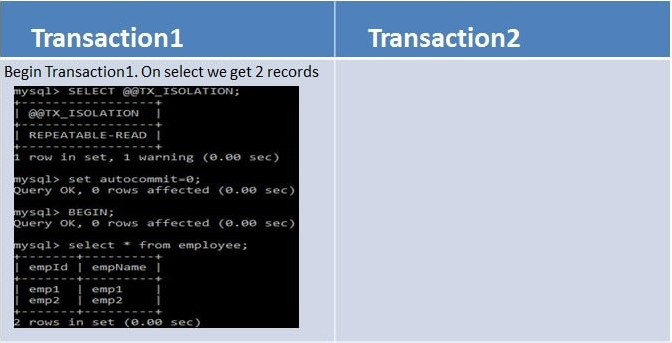
The following are the types of Transaction Isolation Levels-

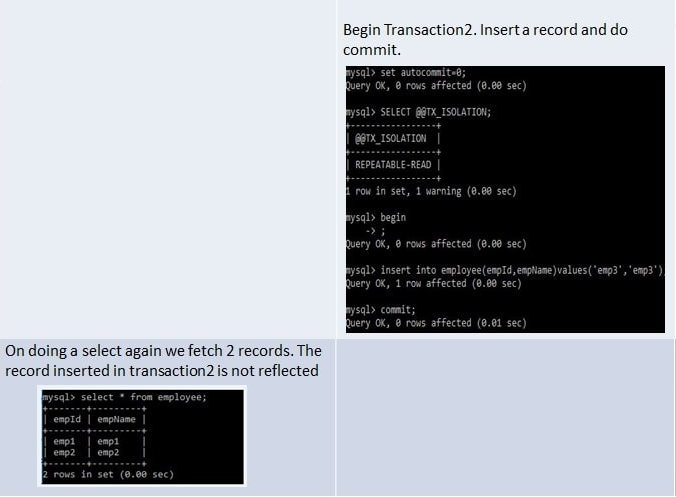
**SERIALIZABLE**  
If two transactions are executing concurrently then it is as if the transactions get executed serially i.e the first transaction gets committed only then the second transaction gets executed. This is total isolation. So, a running transaction is never affected by other transactions. However, this may cause issues as performance will be low and deadlock might occur.



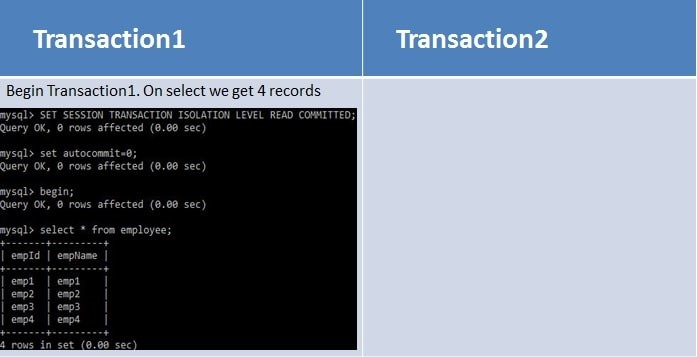


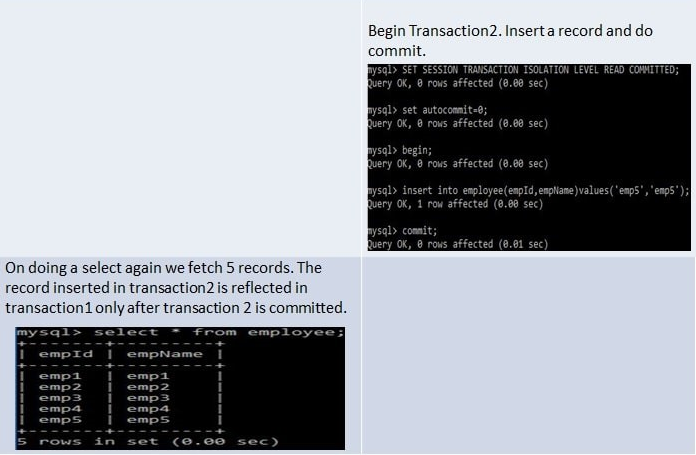
**REPEATABLE\_READ**  
If two transactions are executing concurrently - till the first transaction is committed the existing records cannot be changed by second transaction but new records can be added. After the second transaction is committed, the new added records get reflected in first transaction which is still not committed. For MySQL the default isolation level is REPEATABLE\_READ.  
However the REPEATABLE READ isolation level behaves differently when using mysql. When using MYSQL we are not able to see the newly added records that are committed by the second transaction.





**READ\_COMMITTED**  
If two transactions are executing concurrently - before the first transaction is committed the existing records can be changed as well as new records can be changed by second transaction. After the second transaction is committed, the newly added and also updated records get reflected in first transaction which is still not committed.





**READ\_UNCOMMITTED**  
If two transactions are executing concurrently - before the first transaction is committed the existing records can be changed as well as new records can be changed by second transaction. Even if the second transaction is not committed the newly added andalso updated records get reflected in first transaction which is still not committed.





**Summary**

* **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.
* **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.
* **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.

|  |  |  |  |
| --- | --- | --- | --- |
| **Isolation Level** | **Dirty Reads** | **Non-Repeatable Reads** | **Phantom Reads** |
| **SERIALIZABLE** | This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially | This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially | This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially |
| **REPEATABLE\_READ** | This scenario is not possible as any existing record change gets reflected only if the transaction is committed. So other transaction will never read wrong value. | This scenario is not possible since any record can be changed only after a transaction has been committed. So multiple select statements before transaction commit will always return same existing records. | This scenario is possible as other transactions can insert new records even if first transaction commit has not taken place. |
| **READ\_COMMITTED** | This scenario is not possible as any existing record change gets reflected only if the transaction is committed. So other transaction will never read wrong value. | This scenario is possible as other transactions can modify existing records even if first transaction commit has not taken place. | This scenario is possible as other transactions can insert new records even if first transaction commit has not taken place. |
| **READ\_UNCOMMITTED** | This scenario is possible as any record can be read by other transactions even if the first transaction is not committed. So if first transaction rollbacks the record changes then other transactions will have wrong values | This scenario is possible since any record can be changed even if a transaction is not committed. | This scenario is possible as any record can be inserted even if a transaction is not committed. |

## **Implement Transaction Isolation using Spring Boot**

When using Transaction Isolation with Spring Boot, the default transaction isolation taken is that of the underlying database. So, for our spring boot application the default transaction isolation will be REPEATABLE\_READ since we are using MySQL database. In previous tutorial - [Spring Boot Transaction Management Example](https://www.javainuse.com/spring/boot-transaction) we saw what are transactions and implemented declarative transaction management. We will be modifying this code. We can change the transaction isolation level as follows-

package com.javainuse.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Isolation;

import org.springframework.transaction.annotation.Transactional;

import com.javainuse.model.Employee;

import com.javainuse.model.EmployeeHealthInsurance;

import com.javainuse.service.EmployeeService;

import com.javainuse.service.HealthInsuranceService;

import com.javainuse.service.OrganizationService;

@Service

public class OrganzationServiceImpl implements OrganizationService {

@Autowired

EmployeeService employeeService;

@Autowired

HealthInsuranceService healthInsuranceService;

@Override

// Using Transactional annotation we can define any isolation level supported by the underlying database.

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

public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.insertEmployee(employee);

healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);

}

@Override

@Transactional

public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

}

}