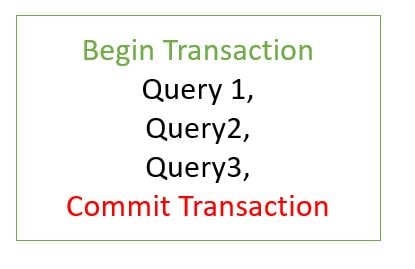
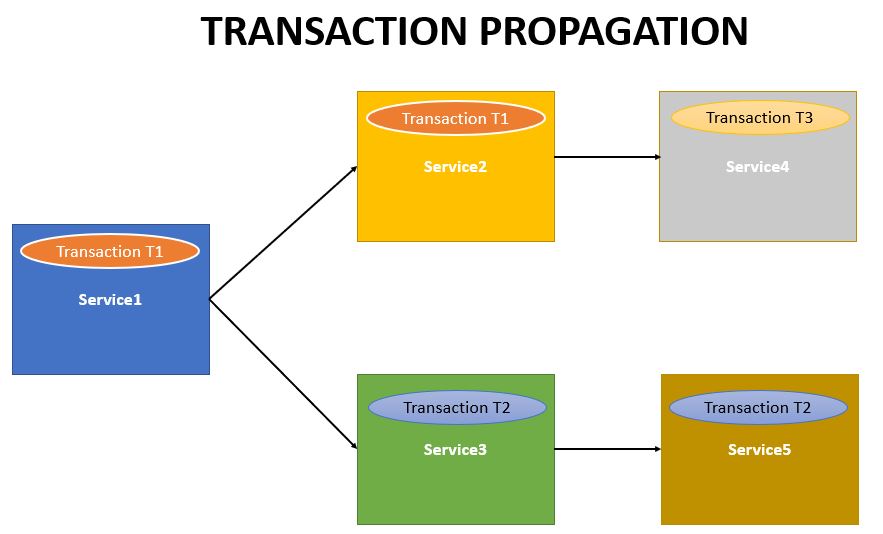
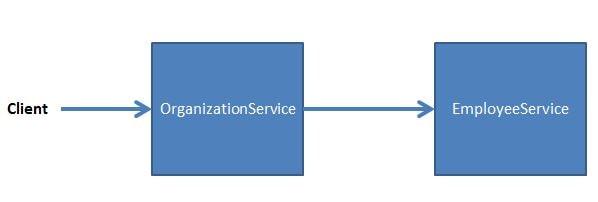
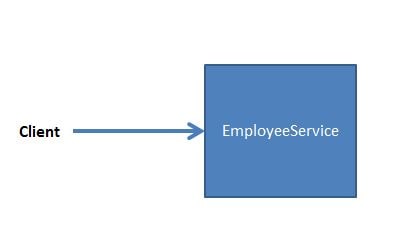
What are Database Transactions?  
A Database transaction is a **single logical unit of work which accesses and possibly modifies the contents of a database.**  
  


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 calling calling component/service already has or does not have a transaction created already.**

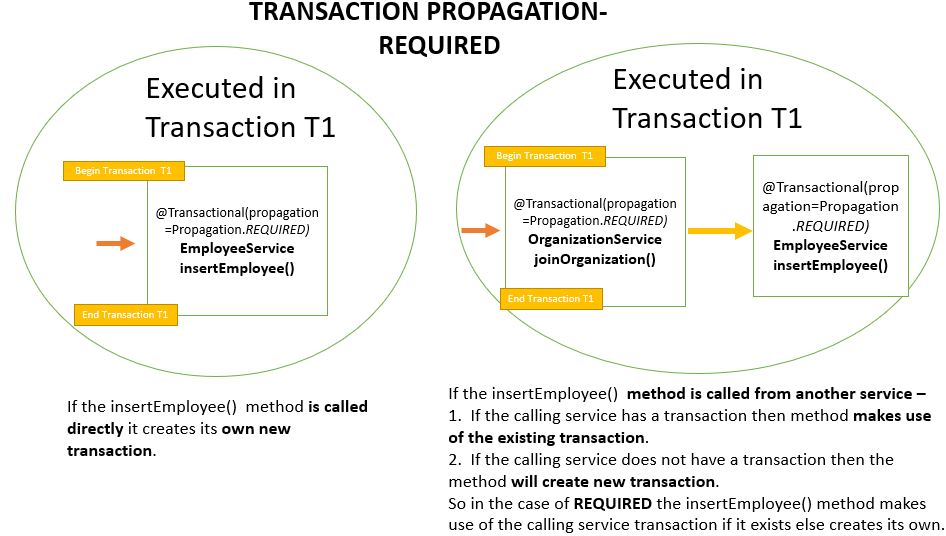


We will be making use of the Spring Boot Transaction Project we developed in previous chapter. It had the Organization service which makes a call to the Employee Service and the Health Insurance Service.  
Also [previous example we had added transaction annotation only to the Organization service.](https://www.javainuse.com/spring/boot-transaction)  
But suppose the user wants to call the Employee Service in both ways i.e.

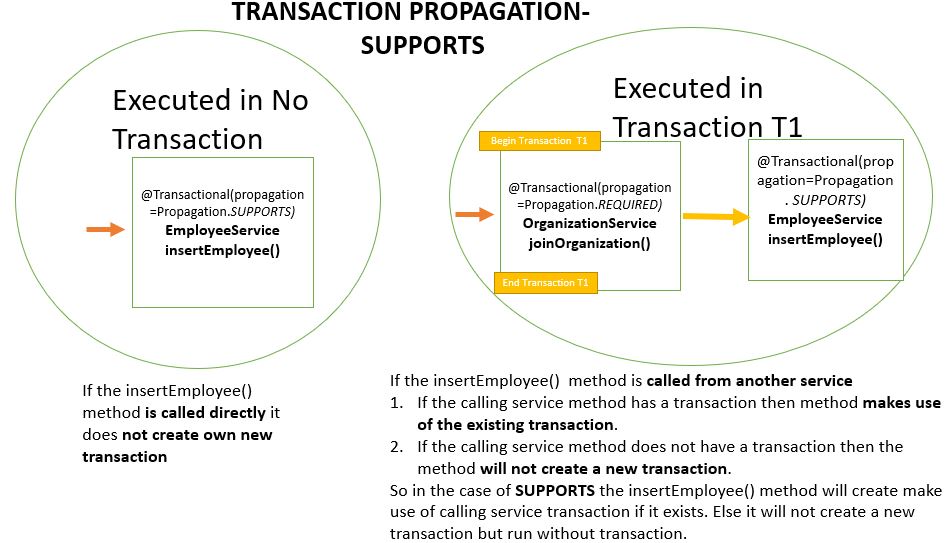
* Call using Organization service  
  

Call the 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.

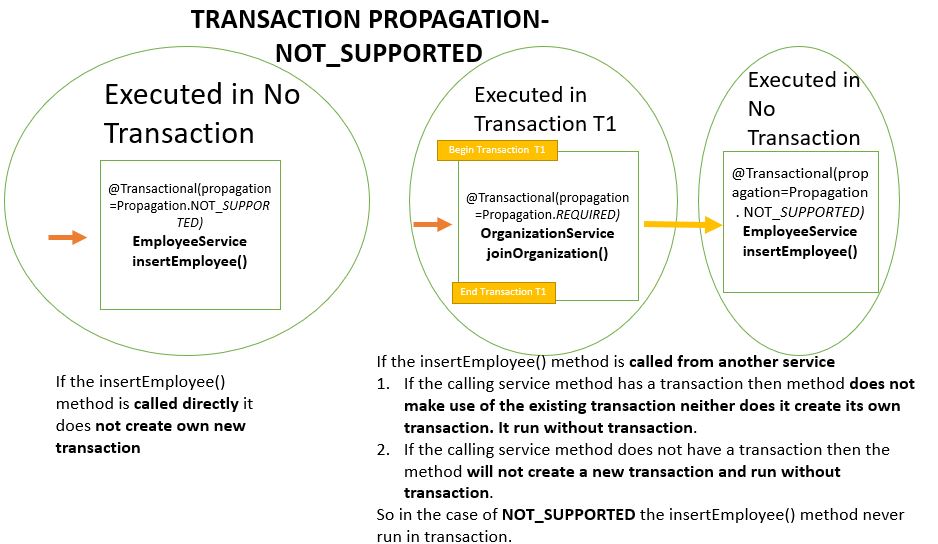
We will be looking at the various propagation scenarios by observing the behaviour of the Organization and Employee service. There are six types of Transaction Propagations-

* **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.

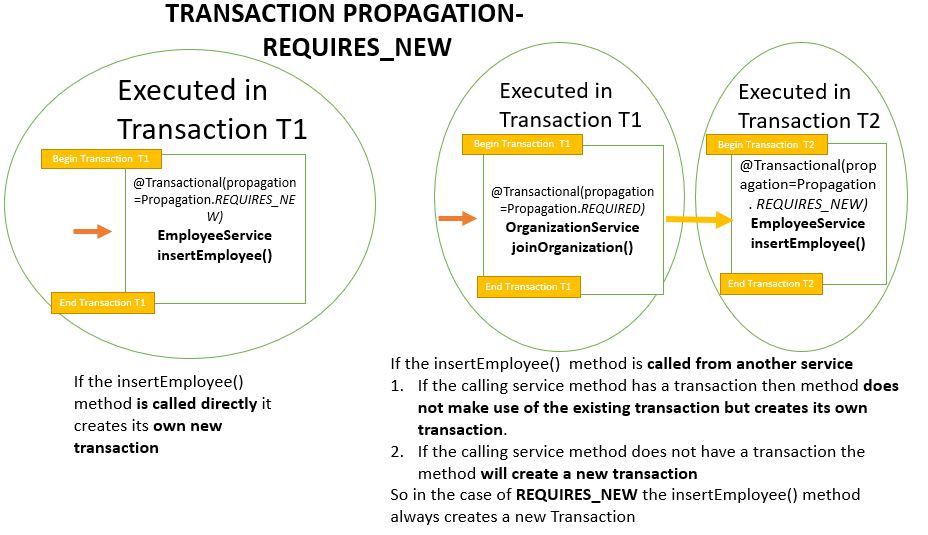
Transaction Propagation - **SUPPORTS**

  
Here both the Organization Service has the transaction propagation defined as **Required** while Employee Service the transaction propagation is defined as **Supports**.

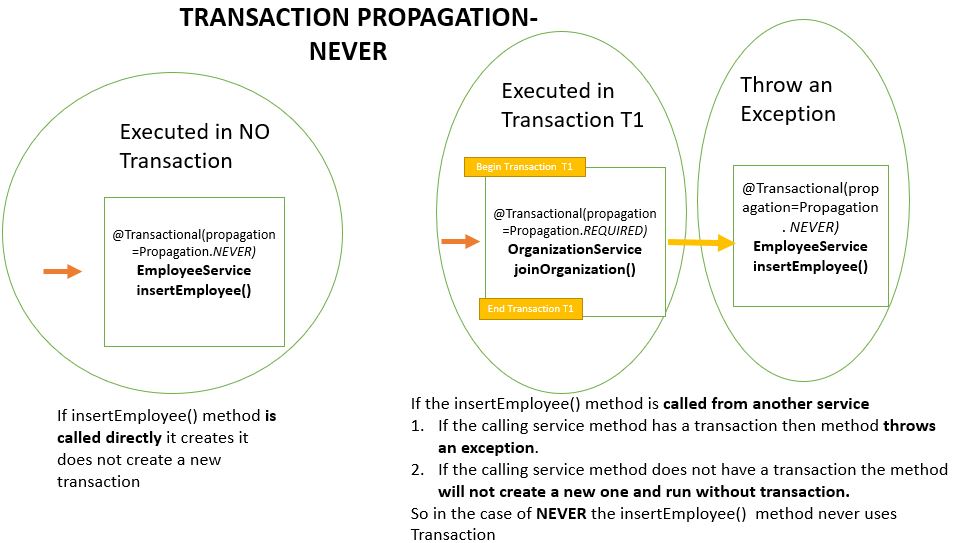
Transaction Propagation - **NOT\_SUPPORTED**

  
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**

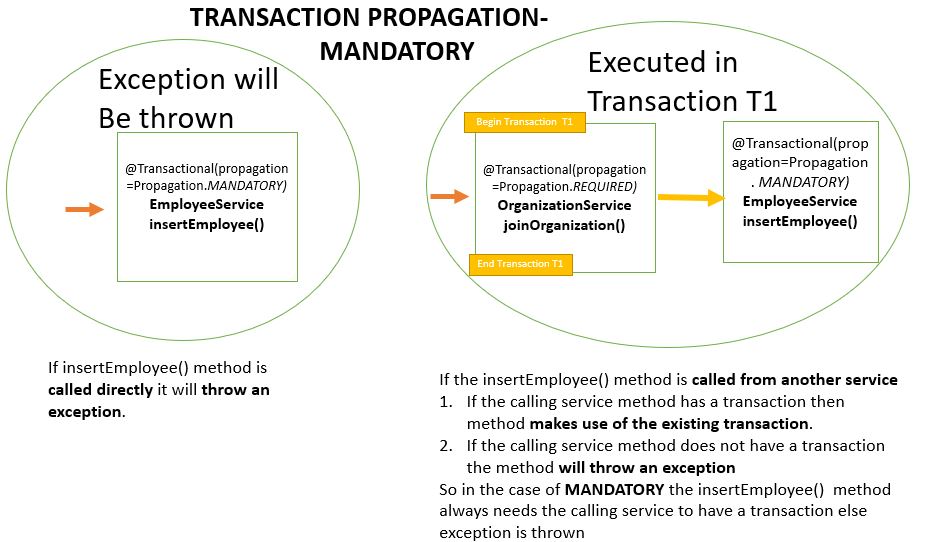
Transaction Propagation - **REQUIRES\_NEW**

  
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**

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 **NEVER**s

## Transaction Propagation - MANDATORY

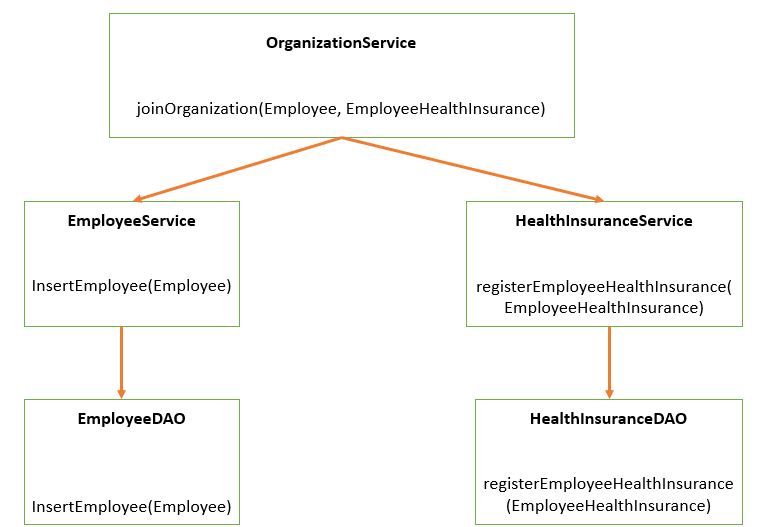
  
Here for the Organization Service we have defined the transaction propagation as **REQUIRED** and the Employee Service have the transaction propagation defined as **MANDATORY**

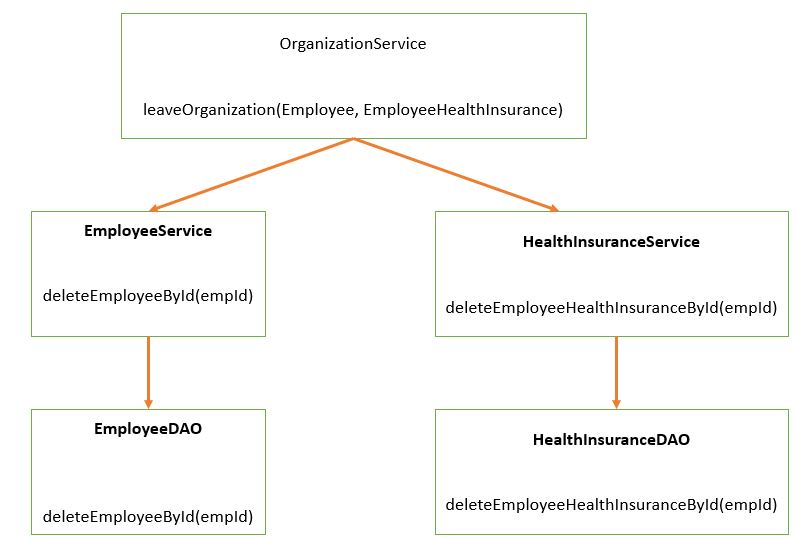
|  |  |
| --- | --- |
| **Propagation** | **Behavior** |
| **REQUIRED** | **Always executes in a transaction.** If there is any existing transaction it uses it. If none exists then only a new one is created |
| **SUPPORTS** | **It may or may not run in a transaction.** If current transaction exists then 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. |

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

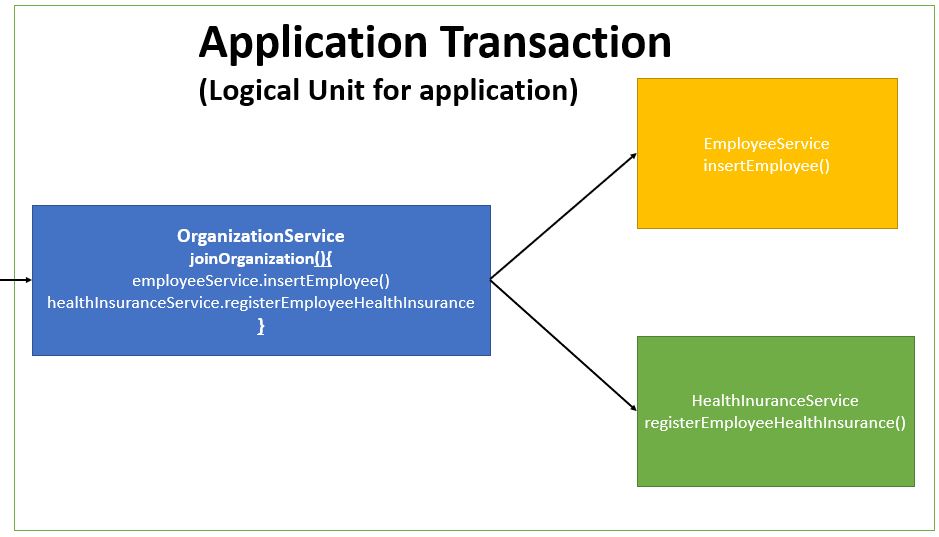
OrganizationService Employee Join Workflow

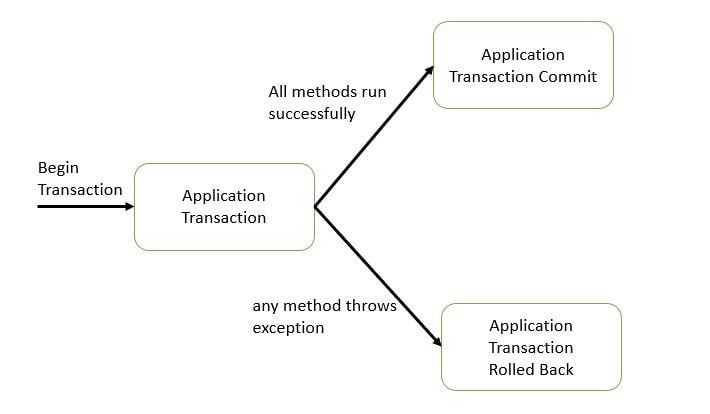


OrganizationService Employee Exit Workflow  


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.

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.



@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

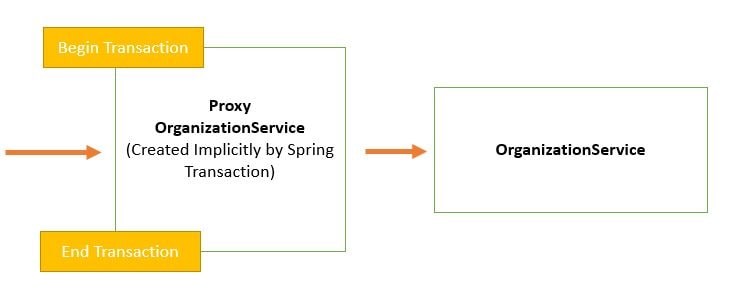
**@Transactional**

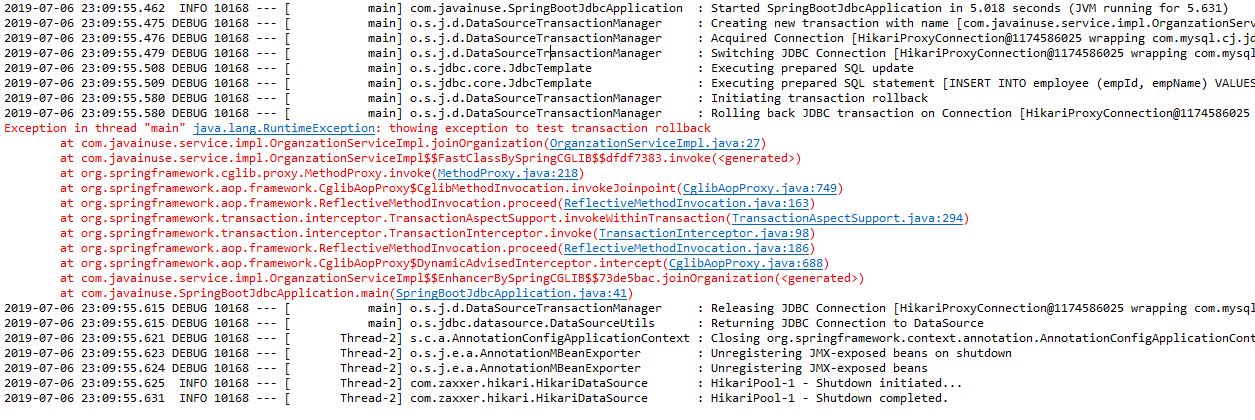
public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {

employeeService.deleteEmployeeById(employee.getEmpId());

healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());

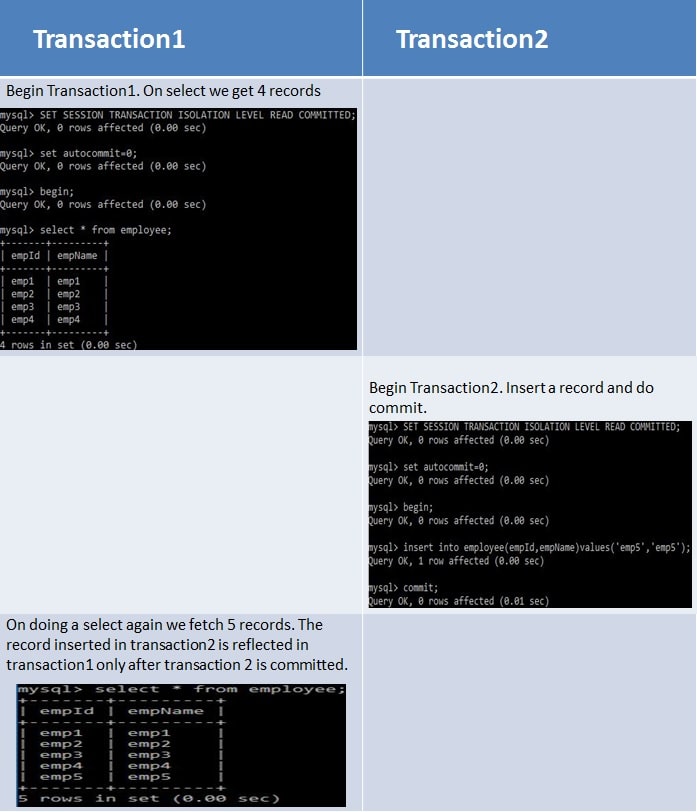
}

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.

**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 **behavior 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 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 and also** updated records get reflected in first transaction which is still not committed.  
  