⏳

Spring Data Jpa

▼ Intro

- Hibernate is a framework in Java which ***comes with an abstraction layer and handles the implementations internally***. The implementations include tasks like writing a query for ***CRUD*** operations or establishing a connection with the databases, etc.
- ***ORM framework*** available for java (***ORM - Object Relational Mapping***)
- It is a technique that maps the object stored in the database. An ORM tool simplifies data creation, manipulation, and access. It internally uses the Java API to interact with the databases.

▼ *JPA*

- Object based Persistence Logic, no queries needed & entire persistence logic written in java.
- since no queries, so it's DB independent.
- Use class names & property names while developing persistent logic.
- ***JPA —>*** *java persistence API*
- JPA is not an implementation. It is only a Java specification.
- JPA explains the handling of data in Java applications.

▼ Advantages

- ***Lightweight and open-source*** – Being lightweight and open-source makes it accessible and efficient.
- ***Increased performance*** – Using cache memory helps in fast performance.
- ***Database Independence*** – Being database-independent gives it the ability to work with different databases.
- ***Auto DDL Operations*** – automatic table creation saves us from manually creating tables.
- It takes care of ***mapping Java classes*** databases ***using XML files*** without writing any code.
- We can ***directly store and retrieve data*** directly from the database using simple APIs.
- It ***does not require*** any application ***server to operate***.
- Minimizes database access with smart fetching strategies.
- It provides ***simple querying*** of data.

▼ Functionalities

- ***Hibernate uses Hibernate Query Language*** which makes it database independent.
- It supports ***auto DDL operations***.
- Hibernate has ***Auto Primary Key Generation*** support.
- It ***supports Cache memory***.
- ***Exception handling*** is ***not mandatory*** for hibernate.

- The most important is ***hibernate is an ORM tool***.

▼ Annotations

ANNOTATION	*USE*
@Id	To metion *primary key* of table.
@Entity @Entity(name = "entity_name") @Table(name = "table_name")	To specify the *class is a table*. To change the *Entity Name*. To change the *Table Name*.
@Column(name = "column_name")	To change the *Column Name.*
@Transient	Used to indicate that a field is not to be persisted in the database.
@OneToOne	To achieve *one-one* relation.
@OneToMany	To achieve *one-many* relation.
@ManyToOne	To achieve *many-one* relation.
@ManyToMany	To achieve *many-many* relation.
@Cachable	To specify the entity is *eligible for caching*.
@Cache	To specify cache strategy.
@GeneratedValue(strategy = GenerationType.IDENTITY)	*auto increment* of primary key
@UpdateTimestamp	Saves timestamp when record updated.
@CreationTimestamp	Saves timestamp when record created.
@Version	Stores the count of record updated. @Version int count;
@Param	gives method parameter a concrete name and bind the name in the query.
@Modifying	Used to enhance the @Query annotation so that we can execute not only SELECT queries, but also INSERT, UPDATE, DELETE, and even DDL queries.
@Transactional	Used when you want the certain method/class(=all methods inside) to be executed in a transaction.
@JoinColumn(name = "column_name", referencedColumnName = "foreign_columnn_name")	Used to specify a column for joining an entity association or element collection
inverseJoinColumn attribute	

▼ Embeddable Object / Component Mapping

- It represents the ***has-a*** relationship.
- The composition is ***stronger association*** where the contained object has ***no existence of its own***.
  - i.e; ***Employee has a Address.*** (***no employee = no address***)
  - since, employee & address are strongly related, its better to store in a single table.
- ***@Embedded*** for component class.
- ***@Embeddable*** for helper class.
- Inserting Object into Another Object
- The class should be annotated with @Embeddable

```
public class App {
    public static void main(String[] args) {
        Configuration con = new Configuration().configure().addAnnotatedClass(Student.class);
        SessionFactory sf = con.buildSessionFactory();
        Session s = sf.openSession();

        StudName sn = new StudName();
        sn.setFname("Gangadhar");
        sn.setLname("Puram");

        Student s1 = new Student();
```

```
        s1.setId(1);
        s1.setMarks(80);
        s1.setName(sn);

        Student s2 = s.get(Student.class, 1);
        System.out.println(s2);
    }
}
```

```
package org.example;

import jakarta.persistence.Embeddable;

@Embeddable
public class StudName {
    private String fname;
    private String lname;

    public String getFname() {
        return fname;
    }

    public void setFname(String fname) {
        this.fname = fname;
    }

    public String getLname() {
        return lname;
    }

    public void setLname(String lname) {
        this.lname = lname;
    }

    @Override
    public String toString() {
        return "StudName{" + "fname='" + fname + '\'' + ", lname='" + lname + '\'' + '}';
    }
}
```

```
package org.example;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import org.example.StudName;

@Entity
class Student {
    @Id
    private int id;
    private int marks;
    @Embedded
    private StudName name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getMarks() {
        return marks;
    }

    public void setMarks(int marks) {
        this.marks = marks;
    }

    public StudName getName() {
        return name;
    }

    public void setName(StudName name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student{" + "id=" + id + ", marks=" + marks
+ ", name=" + name + '}';
    }
}
```

- Student class (var name type is StudName, name can has fname & lname, which is imlpemented using StudName class)
- Thus, Embeddable Object is achieved.
- In table *fname, lname* will be just stored as *additional columns.*

▼ Mapping Relations (cascading has some samples)

▼ One-One

> 💡 One person can only have one aadhar

- ***@OneToOne*** annotation used to achieve this mapping.

```java
package org.example;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.OneToOne;

@Entity
public class Student {
    private String name;
    @Id
    private int id;
    @OneToOne
    private Laptop lap;

    public String getName() {return name;}

    public void setName(String name) {this.name = name;}

    public int getId() {return id;}

    public void setId(int id) {this.id = id;}

    public Laptop getLap() {return lap;}

    public void setLap(Laptop lapId) {this.lap = lapId;}
}
```

```java
package org.example;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class Laptop {
    private String brand;
    @Id
    private int lapId;

    public String getBrand() {return brand;}

    public void setBrand(String brand) {this.brand = brand;}

    public int getLapId() {return lapId;}

    public void setLapId(int lapId) {this.lapId = lapId;}
}
```

// laptop table

LapID	***Brand***
101	Dell
102	Mac

// student table

Id	*Name*	*Lap_LapId*
1	Gangadhar	101
2	Ganesh	102
3	Mani	103

```java
package org.example;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        Laptop l1 = new Laptop();

        l1.setLapId(101);
        l1.setBrand("dell");

        s1.setName("gangadhar");
        s1.setId(1);
        s1.setLap(l1);

        Configuration con = new Configuration().configure().
        SessionFactory sf = con.buildSessionFactory();
        Session s = sf.openSession();

        Transaction t = s.beginTransaction();
        s.save(s1);
        s.save(l1);
        t.commit();
        session.close();
    }
}
```

▼ One-Many / Many-One

💡 One company has many employees.

💡 Many Employees work for a Company

- ***@OneToMany*** annotation used to achieve this mapping.
- ***@ManyToOne*** annotation used to achieve this mapping

```java
package org.example;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.OneToMany;

import java.util.ArrayList;
import java.util.List;
```

```java
package org.example;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToOne;


@Entity
class Laptop {
```

```
@Entity
class Student {
    @Id
    private int sid;
    private String name;
    @OneToMany(mappedBy = "student")
    private List<Laptop> lapList = new ArrayList<>();

    public int getSid() {return sid;}

    public void setSid(int sid) {this.sid = sid;}

    public String getName() {return name;}

    public void setName(String name) {this.name = name;}

    public List<Laptop> getLapList() {
        return lapList;
    }

    public void setLapList(List<Laptop> lapList) {this.lapList = lapList;}
}
```

```
    @Id
    private int lid;
    private String brand;
    @ManyToOne
    private Student student;

    public int getLid() {return lid;}

    public void setLid(int lid) {this.lid = lid;}

    public String getBrand() {return brand;}

    public void setBrand(String brand) {this.brand = brand;}

    public Student getStudent() {return student;}

    public void setStudent(Student student) {this.student =
}
```

// laptop table

LapID	***Brand***	Student_sid
101	dell	1
102	apple	1
103	lenovo	1

// student table

Id	*Name*
1	Gangadhar

```
package org.example;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        Laptop l1 = new Laptop();
        Laptop l2 = new Laptop();
        Laptop l3 = new Laptop();

        l1.setLapId(101);
        l1.setBrand("dell");

        l2.setLapId(102);
        l2.setBrand("apple");

        l3.setLapId(103);
        l3.setBrand("lenovo");

        s1.setName("gangadhar");
        s1.setId(1);
        s1.setLap(l1);
        s1.setLap(l2);
        s1.setLap(l3);

        Configuration con = new Configuration().configure().
        SessionFactory sf = con.buildSessionFactory();
        Session s = sf.openSession();
        Transaction t = s.beginTransaction();

        s.save(s1);
        s.save(l1);
        s.save(l2);
        s.save(l3);

        t.commit();
        session.close();
    }
}
```

▼ Many-Many

> 💡 Many People read Many Books

- *@ManyToMany* annotation used to achieve this mapping

```
package org.example;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToMany;
```

```
package org.example;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToMany;
```

```
import java.util.ArrayList;
import java.util.List;

@Entity
class Student {
    @Id
    private int sid;
    private String name;
    @ManyToMany(mappedBy = "student")
    private List<Laptop> lapList = new ArrayList<>();

    public int getSid() {return sid;}

    public void setSid(int sid) {this.sid = sid;}

    public String getName() {return name;}

    public void setName(String name) {this.name = name;}

    public List<Laptop> getLapList() {return lapList;}

    public void setLapList(List<Laptop> lapList) {this.lapList = lapList;}
}
```

```
import java.util.ArrayList;
import java.util.List;


@Entity
class Laptop {
    @Id
    private int lid;
    private String brand;
    @ManyToMany
    private List<Student> student=new ArrayList<>();

    public int getLid() {return lid;}

    public void setLid(int lid) {this.lid = lid;}

    public String getBrand() {return brand;}

    public void setBrand(String brand) {this.brand = brand;}

    public List<Student> getStudent() {return student;}

    public void setStudent(List<Student> student) {this.stud
}
```

```
package org.example;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

class Main {
    public static void main(String[] args) {

        Configuration con = new Configuration().configure().
                            addAnnotatedClass(Student.class).
                            addAnnotatedClass(Laptop.class);
        SessionFactory sf = con.buildSessionFactory();
        Session session = sf.openSession();

        Student s1 = new Student();
        Student s2 = new Student();

        Laptop l1 = new Laptop();
        Laptop l2 = new Laptop();

        l1.setLid(101);
        l1.setBrand("dell");

        l2.setLid(102);
        l2.setBrand("apple");

        s1.setSid(1);
        s1.setName("n1");

        s2.setSid(2);
        s2.setName("n2");

        l1.getStudent().add(s1);
        l1.getStudent().add(s2);

        l2.getStudent().add(s1);
        l2.getStudent().add(s2);

        Transaction t = session.beginTransaction();
        session.save(l1);
        session.save(l2);
        session.save(s1);
        session.save(s2);
        t.commit();
        session.close();
    }
}
```

// laptop

Lid	*Brand*
101	dell
102	apple

// student

Sid	*Name*
1	n1
1	n2

// laptop_student

LapList_lid	*student_sid*
101	1
101	2
102	1
102	2

▼ Update/Delete Data

```
@Modifying
@Transactional
```

```
@Query("update Student set name = ?1 where email = ?2")
public void updateStud(String firstName, String email){}
```

▼ Cascading

- Without persisting course in DB, we can’t assign a course material for the course. (course is the foreign key of course material)
  - i.e; the course material is dependent on course persistence.
  - In Such scenarios, Cascading comes into play.

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "teacher_id", referencedColumn = "teacherId")
private List<Course> course;
```

```
@ManyToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "teacher_id", referencedColumn = "teacherId")
private Teacher teacher;
```

```
@OneToOne(cascade = CascadeType.ALL)
private Course course;
```

```
@ManyToMany
@JoinTable(name = "student_course",
           joinColumns = @JoinColumn(
                name = "course_id",
                referencedColumnName = "courseId"
           ),
           inverseJoinColumns = @JoinColumn(
                name = "student_id",
                referencedColumnName = "studentId"
           )
)
private List<Student> students;
```

Type	*Use*
Persist	If parent is persisted, then all its related entites also persisted.
Merge	If parent is merged, then all its related entites also merged.
Detach	If parent is detached, then all its related entites also detached.
Refresh	If parent is refreshed, then all its related entites also refreshed.
Remove	If parent is removed, then all its related entites also removed.
All	All above operations can be applied to entities related to parent.

▼ Optionality (entity depending on another entity)

💡 Whenever we are trying to save course, course material is mandatory to save in repo.

```
@OneToOne(optional = false)
private Course course;              // indicates if we try to save course, course material is required
```

▼ Fetch Types

💡 Only for entities, which has mapping relationship.

1. Eager Fetching (brings course data as well while fetching course material.)

2. Lazy Fetching (fetches only course material but not course.)

```
@OneToOne(fetch = FetchType.EAGER)     //default is 'LAZY'
private Course course;
```

▼ Creation & Updation Time stamping

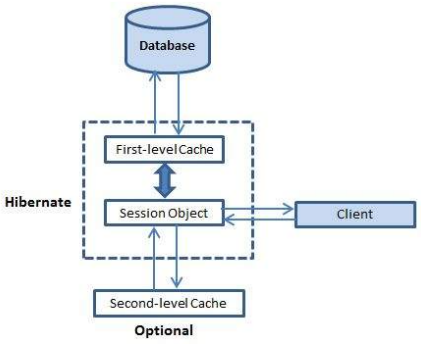
- Keeps track of record saved/inserted.
- Object versioning feature keeps track of record modification.
  - **ex1 :** Track when password changed lastly.
  - **ex2 :** Track last price of stock.
  - **ex3 :** Track last transaction in bank acocunt.
- Annotations used are,
  - ***@CreationTimestamp***
  - ***@UpdateTimestamp***

```
@Version
private int count;                          //stores the count of the record modification.
@CreationTimestamp
private Timestamp creationTime;
@UpdateTimestamp
private Timestamp updationTime;
```

▼ Hibernate Caching

> 💡 When we are accessing ***same data multiple times***, instead of ***calling database*** we use ***Caching***.

- ***Caching*** is a mechanism to enhance the ***performance of a system***. It is a ***buffer memory*** that lies between the ***application and the database***. Cache memory ***stores recently used*** data items in order to ***reduce*** the number of ***database hits*** as much as possible.
- if data is in 1st level cache then it will be used. If not then request goes to 2nd level cache, here also if data not found, the finally the request will be ***sent to database***.
- *click me for caching s/w requirements*



▼ *First Level Cache*

> 💡 If the ***session changes***, the ***cache*** will be ***cleared***.

- ***Session object holds the first level cache data***. It is enabled ***by default***. The first level cache data will ***not*** be ***available to entire application***. An application ***can use many session object***.
- By ***default*** given by ***Hibernate***.

- Used for a ***single Session***.
- If the query is same, then it is default. else we have to set it up.

```
session.beginTranscation();
Emp e1 = s.get(Emp.class, 2);
Emp e2 = s.get(Emp.class, 1);
session.getTransaction().commit();
```

```
session.beginTranscation();
Emp e1 = s.get(Emp.class, 1);            //step 1
Emp e2 = s.get(Emp.class, 1);
session.getTransaction().commit();
```

```
Session s1 = sf.openSession();
s1.beginTransaction();
Emp e1 = s1.get(Emp.class, 2);
Emp e2 = s1.get(Emp.class, 1);
s1.getTransaction().commit();
s1.close();

Session s2 = sf.openSession();
s2.beginTransaction();
Emp e3 = s2.get(Emp.class, 2);
Emp e4 = s2.get(Emp.class, 1);
s2.getTransaction().commit();
s2.close();


//output

Hibernate: select e1_0.empId,c1_0.name,c1_0.ceo,e1_0.empName from Emp e1_0 left join Company c1_0 on c1_0.name=e1_0.company
Hibernate: select e1_0.empId,c1_0.name,c1_0.ceo,e1_0.empName from Emp e1_0 left join Company c1_0 on c1_0.name=e1_0.company
Hibernate: select e1_0.empId,c1_0.name,c1_0.ceo,e1_0.empName from Emp e1_0 left join Company c1_0 on c1_0.name=e1_0.company
Hibernate: select e1_0.empId,c1_0.name,c1_0.ceo,e1_0.empName from Emp e1_0 left join Company c1_0 on c1_0.name=e1_0.company
```

▼ *Second Level Cache*

💡 If there are ***more than one sessions*** need same data, then ***Second level cache used***

💡 **<property name= "hibernate.cache.use_second_level_cache" > *true* </property>**
<property name= "hibernate.cache.region.factory_class" >
org.hibernate.cache.ehcache.internal.EhcacheRegionFactory **</property>**

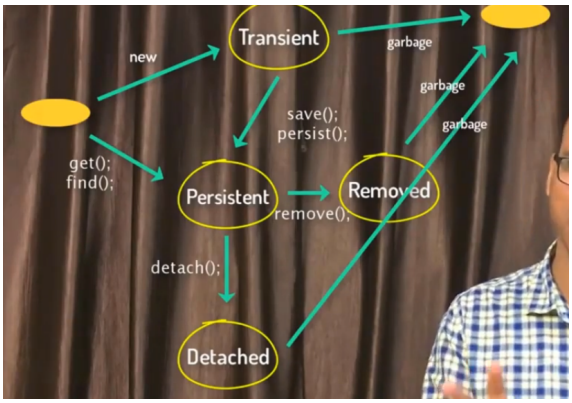
- ***SessionFactory object*** holds the ***second level cache data***. The ***data stored*** in the second level cache will be ***available to entire application***. But we need to ***enable it explicitely***.
- Also called as ***Global Cache***.
- ***One*** per each ***SessionFactory*** object. (***i.e; one per each DB s/w***)
- We have to ***use Third-party library*** to ***achieve*** this. some are,
  - ***EH (****Easy Hibernate****) Cache***
  - ***Swarm Cache***
  - ***OS Cache***
  - ***JBoss Cache***
- we must specify the ***caching strategy,***
  - ***read-only*** : caching will work only for read only.
  - ***nonstrict-read-write*** : works for read & write but one at time.
  - ***read-write*** : works for read & write simultaneosly.
  - ***trasactional*** : works only for transaction.
- Annotations used are,
  - ***@Cacheable***
  - ***@Cache*** (***usage = CacheConcurrencyStrategy.READ_ONLY***)

- Stable Versions,
  - **hibernate core** - 5.6.8
  - **hibernate-ehcache** - 5.6.8
  - **Use javax persistence**
- `session.evict(object)` — clears L1 cache.
- `session.clear()` — clears L2 cache.
- `sessionfactory.getCache().evictAll() ;` — clears L2 cache.

▼ *Hibernate Caching Level 2 with Query*

▼ Hibernate Object States/Persistence Life Cycle

- By default Object state is Transient in Hibernate.
- We can change its state by `session.save(), session.update(), session.persist()……..`
- Object States in Hibernate,
  - ***Transient*** - *default state*
  - ***Persistent*** - *save(), persist(), get(), find()*
  - ***Detached*** - *detach()*
  - ***Removed*** - *remove()*




```
s.beginTransaction();
  s.remove(f);
  s.persist(f);
s.getTransaction().commit();
  s.detach(f);
s.close();
```

▼ SQL in Hibernate / Native SQL

> 💡 We can use Pure SQL in hibernate

Named SQL Queries

- used to perform insert operation.
- direct communication with DB. Hence, good performance

Native SQL Scalar Queries

- Types of params supported,
  - JPA style param (***?1, ?2, ?3***)
  - JDBC style param (***?, ? ,?***)
  - named Param (***:name1, :name2***)

▼ Get vs Load

Get	*Load*
Get returns *actual object*.	Load returns *proxy object*.
Mostly used method.	Rarely used method.
Used to Fetch data.	Used to pass fake object inside other object.
When data isn't present throws *NullPointerException*.	When data isn't present throws *ObjectNotFoundException*.

▼ Uni & Bi-directional Relationship

💡 A unidirectional relationship is ***valid in only one direction***

💡 A bi-directional relationship is ***valid in both directions***

```
@OneToOne(mappedBy = "course")
private CourseMaterial courseMaterial;
```

```
@OneToOne(cascade = CascadeType.ALL)
private Course course;
```

▼ Paging & Sorting

```
public void fun(){
  Pageable firstPageWithThreeRecords = PageRequest.of(0,3);
  Pageable secondPageWithTwoRecords = PageRequest.of(1,2);

  List<Course> courses = courseRepo.findAll(firstPageWithThreeRecords).getContent();
  Long totalElements = courseRepo.findAll(firstPageWithThreeRecords).getTotalElements();
  Long totalPages = courseRepo.findAll(firstPageWithThreeRecords).getTotalPages();
}
```

```
public void fun(){
  Pageable sortByTitle = PageRequest.of(0,3,sort.by("title").descending());
  Pageable sortByTitleAndCredit = PageRequest.of(0,3,sort.by("title").descending().and(sort.by("credit")));

  List<Course> courses = courseRepo.findAll(soortByTitle).getContent();
}
```

▼ Real Time Coding Standards

```
@Entity
@Table(name = "student",
       uniqueConstraints = @UniqueConstraints(
            name = "emailid_unique",
            columnNames = "email_address"
       )
)
public class Student{

  @Id
  @SequenceGenerator(
      name = "student_sequence",
      sequenceName = "student_sequence",
      allocationSize = 1
  )
  @GeneratedValue(
      strategy = GenerationType.SEQUENCE,
      generator = "student_sequence"
  )
  private int id;

  column(name = "student_name")
  private String name;

  column(name = "email_address",
         nullable = false
  )
  private String email;
```

```
@Embeddable
@AttributeOverrides({
    @AttributeOverride(
        name = "name",
        column = @Column(name = "guardian_name")
    ),
    @AttributeOverride(
        name = "email",
        column = @Column(name = "guardian_email")
    )
})
public class Guardian{
  private String name;
  private String email;
}
```

```
    @Embedded
    private Guardian guardian;
}
```

Interview Questions

Configuration

- It is a class in org.hibernate.cfg
- Activates hibernate framework.
- Reads both configuration & mapping files.
- Checks whether config file is correct or not.
    - if correct, returns meta-data to object to represent config file.

SessionFactory

- It is an interface in `org.hibernate`
- Used to create `Session` object.
- Immutable & thread-safe in nature.
- `buildSessionFactory()` gathers metadata in `Configuration` object
- From configuration object, it takes JDBC info & creates connection to DB.

Session

- It is a interface in org.hibernate
- Session object is created based upon SessionFactory object.
- Opens connection/session with database through hibernate.
- Light-Weight object & not thread-safe.
- Used to perform CRUD operations.

Transaction

> 💡 Used only when we need to manipulate data in database.

- It is an interface in org.hibernate
- Used whenever we perform operation in database.
- Instructs DB to make the changes permanent using `commit()`

Points To Remember

- It is not advisable to use `hibernate.ddl-auto=update` in production.
- JPQL Queries are based on the class we created but not on the table.
- Always use wrapper classes for entity properties.
- for Native Queries,

```
@Query(value = "select * from student where email = ?1", nativeQuery = true)
public Student getStudent(String email);

@Query(value = "select * from student where email :emailId", nativeQuery = true)
public Student getStudent(@Param("emailId") String email);
```

- *@Transactional should be used in service layer.*
    - ***example :***

      Let's assume user `A` wants to transfer 100$ to user `B`. What happens is:

      1. We decrease A's account by 100$

2. We add 100$ to B's account

   Let's assume the exception is thrown after succeeding `1)` and before executing `2)`. Now we would have some kind of inconsistency because `A` lost 100$ while `B` got nothing. Transactions means all or nothing. If there is an exception thrown somewhere in the method, changes are not persisted in the database. Something called `rollback` happens.

   If you don't specify `@Transactional`, each DB call will be in a different transaction.

- By default, the fetch is ***Left Outer Join***.