

```

        Hibernate With Jpa
-----JDBC VS JPA-----
JDBC
|
JPA
1.We write the query
| JPA take care of query formation
2.Map the query params with instance variables of class
table (with class) and column names with instance variable
|Map the
----- JPA -----
@Entity -used to declare that a particular class is an entity and its
life cycle is managed by entity manager
@Table - Optional ..should be added if class name does not match with
table name
@Column- Optional ..used to match column name with variable name in class
@Id - represents primary key
@GeneratedValue - auto increment of primary key

```

STEP - 1:

Dependencies => jpa,mysql,web

DataBase Queries :

```

        create database hibernate;
        use hibernate;

        create table person(
            id int auto_increment primary key,      ***** Focus as it is
auto_increment *****
            name varchar(255),
            location varchar(255)
        );

```

STEP -2.Application.properties  
server.port=1000

```

spring.datasource.url=jdbc:mysql://localhost:3306/hibernate
spring.datasource.username=root
spring.datasource.password=root

spring.jpa.show-sql=true

```

Note :spring.jpa.show-sql=true is used to see sql queries executed.Create an Entity

STEP -3:

```

@Entity
//@Table(name = "person")
@NamedQuery(name="find_all_persons",query = "select p from Person p")
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)      -- Iske
alawa koi or use krte h toh error aa rha h
    private int id;
    //@Column(name = "name") --- Optional ..used to match column name
with variable name in class
    private String name;

```

```

    private String location;

    ** Costructor where id is not present
}

```

Note : 1.Create a constructor where id is not present  
 2.Use auto\_increment in table and IDENTITY strategy in Class for Primary key

STEP -4.Create a Repository using @Repository  
 -Add @Transactional  
 -Connect to database using PersistenceContext and EntityManager  
 -To Retrieve data From Table using primary key we will use  
`EntityClass find(EntityClass, Primary Key);`  
 -To Perform Insert or Update we use ..if new entry is there then it is added otherwise updated if exists  
`EntityClass merge(Entity);`  
`void persist(Entity);`  
 -To Perform Delete we use  
`void remove(Entity)`  
 -To Perfrom findAll  
`List<ResultClass> createNamedQuery(String QueryName,ResultClass).getResultList()`  
`Note : we have added NamedQuery annotation in Entity Class`

```

@Repository
@Transactional
public class PersonJpaDao {

    //connect to database
    @PersistenceContext
    EntityManager entityManager;

    ** Reference : @NamedQuery(name="find_all_persons",query = "select p from Person p") **
    public List<Person> findAll(){
        TypedQuery<Person>
        namedQuery=entityManager.createNamedQuery("find_all_persons",
        Person.class);
        return namedQuery.getResultList();
    }

    public Person findById(int id) {
        return entityManager.find(Person.class, id);
        //entityManager.find(EntityClass, Primary Key);
    }

    public Person updateOrInsert(Person person) {
        return entityManager.merge(person);
    }

    public void deleteById(int id) {
        Person p=findById(id);
        entityManager.remove(p);
    }
}

```

```
}
```

#### 4. SpringBootApplication class

```
@SpringBootApplication
public class JPAdemoApplication implements CommandLineRunner{

    private Logger logger=LoggerFactory.getLogger(this.getClass());

    @Autowired
    PersonJpaDao dao;

    public static void main(String[] args) {
        SpringApplication.run(JPAdemoApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        // TODO Auto-generated method stub
        logger.info("JPA User Id 1001->{}",dao.findById(1001));
        logger.info("JPA Insert new Sequential User id-
>{}",dao.updateOrInsert(new Person("Pawan", "Kota")));
        logger.info("JPA Update User id 1002-
>{}",dao.updateOrInsert(new Person(1002,"Pawan", "Kota-Ghar")));
        dao.deleteById(1001);
        logger.info("JPA findAll ->{}",dao.findAll());
    }
}

-----persist() vs merge()-----
-----
1.Scenario-1
MyEntity e = new MyEntity();

--> tran starts
em.persist(e);
e.setSomeField(someValue);
--> tran ends, and the row for someField is updated in the database

2.Scenario-2
// tran starts
MyEntity e = new MyEntity();
em.merge(e);
e.setSomeField(anotherValue);
// tran ends but the row for someField is not updated in the database
// (you made the changes *after* merging)

3.Scenario-3
// tran starts
MyEntity e = new MyEntity();
MyEntity e2 = em.merge(e);
e2.setSomeField(anotherValue);
// tran ends and the row for someField is updated
// (the changes were made to e2, not e)

persist:
    Insert a new entry to the database
```

Attach the object to the Persistence context.

**merge:**

- Find an attached object with the same id and update it.
- If exists update and return the already attached object.
- If doesn't exist insert the new register to the database.

-----flush(), detach() ,clear()-----

flush() -> Changes upto that point are saved to database

detach(entity) -> Remove the given entity from the persistence context

clear() -> Clear the persistence context, Remove all the given entity from the persistence context

refresh()->is to refresh the state of an instance from the database

**Scenario 1:**

```
public void PlayWithEntityManager() {
    Course course1=new Course("Course JPA");
    em.persist(course1);
    course1.setName("Course JPA Updated");

    Course course2=new Course("Course Angular");
    em.persist(course2);
    course2.setName("Course Angular Updated");
}
```

**Console Output :**

```
Hibernate: insert into course (name) values (?)
Hibernate: insert into course (name) values (?)
Hibernate: update course set name=? where id=?
Hibernate: update course set name=? where id=?
```

**Scenario 2:**

```
public void PlayWithEntityManager() {
    Course course1=new Course("Course JPA");
    em.persist(course1);
    em.flush();
    course1.setName("Course JPA Updated");
    em.flush();

    Course course2=new Course("Course Angular");
    em.persist(course2);
    em.flush();
    course2.setName("Course Angular Updated");
    em.flush();
}
```

**Console Output:**

```
Hibernate: insert into course (name) values (?)
Hibernate: update course set name=? where id=?
Hibernate: insert into course (name) values (?)
Hibernate: update course set name=? where id=?
```

**Scenario 3:**

```
public void PlayWithEntityManager() {
    Course course1=new Course("Course JPA");
    em.persist(course1);
    Course course2=new Course("Course Angular");
    em.persist(course2);
```

```

        em.detach(course2);
        course1.setName("Course JPA Updated");
        course2.setName("Course Angular Updated");
    }
}

```

Console Output :

```

Hibernate: insert into course (name) values (?)
Hibernate: insert into course (name) values (?)
Hibernate: update course set name=? where id=?

```

Scenario 4:

```

public void PlayWithEntityManager() {
    Course course1=new Course("Course JPA");
    em.persist(course1);
    Course course2=new Course("Course Angular");
    em.persist(course2);
    em.clear();
    course1.setName("Course JPA Updated");
    course2.setName("Course Angular Updated");
}

```

Console Output :

```

Hibernate: insert into course (name) values (?)
Hibernate: insert into course (name) values (?)

```

Scenario 5 :

```

public void PlayWithEntityManager() {
    Course course1=new Course("Course JPA");
    em.persist(course1);           --> It will only assign auto
                                     gerenerated id here and no db transaction happens
    Course course2=new Course("Course Angular");
    em.persist(course2);           --> It will only assign auto
                                     gerenerated id here and no db transaction happens
    em.flush();                   --> Db transaction happens here

    course1.setName("Course JPA Updated"); --> No Db Transaction ..just
change in object value happens
    course2.setName("Course Angular Updated"); --> No Db
Transaction ..just change in object value happens

    em.refresh(course1); --> select query executed to refresh course 1
to sync with database record
}

```

Console Output:

```

Hibernate: insert into course (name) values (?)
Hibernate: insert into course (name) values (?)
Hibernate: select course0_.id as id1_0_0_, course0_.name as name2_0_0_
from course course0_ where course0_.id=?
Hibernate: update course set name=? where id=?

```

-----JPQL (Java Persistence Query Language )-----

```

public List<Course> playWithJPQLQuery() {
    Query query=em.createQuery("select c from Course c")
    List courses=query.getResultList();
    return courses;
}

```

```

        or
TypedQuery tQuery=em.createQuery("select c from Course
c",Course.class)
List<Course> courses=tQuery.getResultList();
return courses;
}
Note : Select c from Course c .....here Course is class name /entity
name
select c from Course c where name like '%happy%'

-----@CreationTimestamp and @UpdateTimestamp -----
-----

create table course (
    id int auto_increment primary key,
    name varchar(255),
    created_ts timestamp,
    updated_ts timestamp
);

@Entity
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    @CreationTimestamp
    private LocalDateTime createdTs;
    @UpdateTimestamp
    private LocalDateTime updatedTs;

    //no args constructor
    //args constructor
}

@Repository
@Transactional
public class CourseRepository {

    @PersistenceContext
    EntityManager em;

    public Course insertPlay(Course c) {
        em.persist(c);
        return c;
    }
}

@SpringBootApplication
public class HibdemoApplication {

    private Logger logger=LoggerFactory.getLogger(this.getClass());

    @Autowired
    CourseRepository crepo;
}

```

```

        public static void main(String[] args) {
            SpringApplication.run(HibdemoApplication.class, args);
            logger.info("courses->{}", crepo.insertPlay(new Course("new
kjashdk")));
        }
    }

```

-----@NamedQuery and @NamedQueries -----  
Note : we have seen example of @NamedQuery above

Code Example of @NamedQueries :

```

@Entity
@NamedQueries(value = {
    @NamedQuery(name = "query_name_1", query = "select c from Course c"),
    @NamedQuery(name = "query_name_2", query = "select c from Course c where
name like '%Happy%'")
})
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    @CreationTimestamp
    private LocalDateTime createdTs;
    @UpdateTimestamp
    private LocalDateTime updatedTs;

}

@Repository
@Transactional
public class CourseRepository {

    @PersistenceContext
    EntityManager em;

    public List<Course> playWithNamedQueries() {
        TypedQuery<Course> tQuery=em.createNamedQuery("query_name_1",
Course.class);
        List<Course> liCourses=tQuery.getResultList();
        return liCourses;
    }
}

Main Class{
@Override
    public void run(String... args) throws Exception {
        // TODO Auto-generated method stub
        logger.info("courses->{}", crepo.playWithNamedQueries());
    }
}

```

```

-----NativeQueries -----
Native Queries are Sql Queries
Note : Native Queries are used when we want to go for mass update ....
read from intenet also

@Repository
@Transactional
public class CourseRepository {

    Logger logger=LoggerFactory.getLogger(this.getClass());

    @PersistenceContext
    EntityManager em;

    public void nativeQueryBasic() {
        Query query=em.createNativeQuery("select * from
Course",Course.class);
        List<Course> courses=query.getResultList();
        logger.info("select * from Course ->{}",courses);
    }

    public void nativeQueryWithParam() {
        Query query=em.createNativeQuery("select * from Course where
id=?",Course.class);
        query.setParameter(1, 1);
        List<Course> courses=query.getResultList();
        logger.info("select * from Course where id=? ->{}",courses);
    }

    public void nativeQueryWithNamedParams() {
        Query query=em.createNativeQuery("select * from Course where
id=:id",Course.class);
        query.setParameter("id", 1);
        List<Course> courses=query.getResultList();
        logger.info("select * from Course where id=:id -
>{}",courses);
    }

    public void nativeQueryUpdate() {
        Query query=em.createNativeQuery("update Course set
name=?",Course.class);
        query.setParameter(1, "pawan");
        int rows=query.executeUpdate(); --> work for update and
delete query
        logger.info("update Course set name=? ->{}",rows);
    }
}

-----One to One Mapping -----

create table passport(
    id int auto_increment primary key,
    number varchar(255)
);

create table Student(

```

```

id int auto_increment primary key,
name varchar(255),
passport_id int ,
FOREIGN KEY (passport_id) REFERENCES passport(id)
)

@Entity
public class Passport {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String number;

}

@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

    @OneToOne //(fetch = FetchType.LAZY) --> By default fetch type is
EAGER
    private Passport passport;

}

@Repository
@Transactional
public class StudentRepository {

    @PersistenceContext
    EntityManager em;

    public void insertStudentWithPassport() {
        Passport passport =new Passport("P123456");
        em.persist(passport);

        Student student=new Student("Pawan");
        student.setPassport(passport);
        em.persist(student);
    }

    //Default Fetch is Eager Fetching .. if we retrieve Student
..Passport will also be fetched
    public void retrieveStudentAndPassport() {
        int studentPrimaryKeyId=1;
        Student student=em.find(Student.class, studentPrimaryKeyId);
        System.out.println(student); // Student [id=1, name=Pawan,
passport=Passport [id=1, number=P123456]]
        Passport passport=student.getPassport();
    }
}

```

```

        passport.setNumber("pssport Number Updated by detching
Student");
    }

}

```

Note : Wrong Approach

```

public void insertStudentWithPassport() {
    Passport passport =new Passport("P123456");

    Student student=new Student("Pawan");
    student.setPassport(passport);
    em.persist(student);
}

```

----- Entity Manager ,Persistence Context and  
Transactional -----

Entity Manager :

1.Entity Manager is an interface to something called PersistenceContext  
2.Entity Manager helps to bind entity to PersistenceContext  
In above cases course1 and course2 are entities and with help of  
EntityManager ..these Entity are stored in PersistenceContext

PersistenceContext :

1.PersistenceContext keeps track of changes in entities during a  
transaction  
2.PersistenceContext also keeps track of changes need to be stored in DB  
3.Transaction and PersistenceContext is limited to that method and and  
operation is executed on Database at end of each method.

```

@Repository
@Transactional
public class StudentRepository {

    @PersistenceContext
    EntityManager em;

    public void methodToUnderstandPersistenceContext() {
        int studentPrimaryKeyId=1;

        //Database Operation 1 - retrieve Student
        Student student=em.find(Student.class, studentPrimaryKeyId);
        //Peristence Context(student)

        //Database Operation 2 - retrieve Passport
        Passport passport=student.getPassport();
        //Peristence Context(student,passport)

        //Database Operation 3 - update passport
        passport.setNumber("K234789");
        //Peristence Context(student,passport++)

        //Database Operation 4 -update student
        student.setName("Kim");
    }
}

```

```

        //Persistence Context(student++,passport++)
    }

}
-----Bi-Directional Relationship -----

```

In Above Case Student is Owning Side of Relationship as Student contains Passport (Acc. to DB and Entity Structure)

But Now we want Passport to contain Student also ... But we have to add Student\_id column in Passport Table

Student table contains Passport\_id and Passport will contain student\_id ....duplication of data will be there ...THIS IS WRONG !!!

To Avoid this we will use

mappedBy property of @OneToOne

```

@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

```

```

    @OneToOne
    private Passport passport;

```

```

    // No args constructor
    //args constructor without id
    //getters and setters
}

```

```

@Entity
public class Passport {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String number;

```

```

    @OneToOne(mappedBy = "passport")
    private Student student;
}

```

```

@Repository
@Transactional
public class StudentRepository {

```

```

    @PersistenceContext
    EntityManager em;

```

```

    public void fetchPassportAndAssociatedStudent() {
        int passportPrimaryKey=1;
    }
}
```

```

        Passport passport=em.find(Passport.class,
passportPrimaryKey);
        Student student=passport.getStudent();
        System.out.println("passport ->" + passport); // passport -
>Passport [id=1, number=P123456]
        System.out.println("student ->" + student); // student -
>Student [id=1, name=Pawan, passport=Passport [id=1, number=P123456]]
    }

deleteOnlyPassport() {

    //Method - 1
    Passport passport=em.find(Passport.class,1);
    Student student=em.find(Student.class,1);
    student.setPassport(null);
    em.remove(passport);

    //Method - 2
    Student student=em.find(Student.class, 1);
    Passport passport=student.getPassport();
    student.setPassport(null);
    em.remove(passport);
}

}
-----FAQ-----

```

1. When Does Hibernate Sends updates to DataBase ?

Ans)

Scenario 1:

```

@Transactional
void someMethodWithChange() {
    #Line 1
    //Create Objects
    em.persist(user1); --> Auto Generated Id is assigned to user
    em.persist(user2); --> Auto Generated Id is assigned to user

    //change user1
    //change user2
} #Line 2
In Console Output --> 2 insert and then 2 update queries are visible at
#Line 2
Note : PersistenceContext starts at #Line 1 and ends at #Line 2 and
Changes are send to Db at #Line2

```

Scenario 2:

```

@Transactional
void someMethodWithChange() {

    //Create Objects
    em.persist(user1); --> Auto Generated Id is assigned to user
    em.persist(user2); --> Auto Generated Id is assigned to user
    em.flush(); --> Here Changes are Moved to Db...2 Insert queries are
executed

    //change user1
    //change user2
}

```

```
} Again Changes are Moved to DB here... 2 update queries are executed
```

2.Do Read Only Methods need a transaction ?

Ans) No Transaction added

Refer Below Example ..Student has a passport

```
//@Transactional
public void someOperation(){

    int studentPrimaryKeyId=1;

        Student student=em.find(Student.class, studentPrimaryKeyId); #Line1

        Passport passport1=em.find(Passport.class, 1); #Line2

        Passport passport=student.getPassport(); #Line3

        passport.setNumber("K234789"); #Line4

        student.setName("Kim"); #Line5

}
```

When we start the application we can see #Line1 and #Line2 executed .... bcoz these lines has entity manager(em) and each em has associated default transaction.....from #Line3 ..execution fails bcoz Fetch type we have set as lazy and Student gets loaded on # Line1 but its related passport is not fetched ...#Line3 Tries to fetch passport but we need PersistenceContext connection along with transaction to execute Db Query to fetch passport details

Agar Yaha Eager Loading ho to there is no problem while start of application and all will execute

Note : Each em has associated Default transactional so it will work even if method or class level transactional is not available

-----ManToOne or OneToMany-----

By Default Fetch Strategy on @OneToMany Side of Relationship is Lazy  
(Step-33 by Ranga on Udemy)

By Default Fetch Strategy on @ManyToOne Side of Relationship is Eager  
(Step-33 by Ranga on Udemy)

```
create table course (
id int auto_increment primary key,
name varchar(255)
);

create table review(
id int auto_increment primary key,
rating varchar(255),
description varchar(255),
course_id int ,
FOREIGN KEY (course_id) REFERENCES course(id)
);
```

```

insert into course values(1001,'course1');
insert into course values(1002,'course2');
insert into review values(2001,'5-star','wonderful',1001);
insert into review values(2002,'4-star','excellent',1001);

@Entity
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    @OneToMany(mappedBy = "course")
    private List<Review> reviews=new ArrayList<>();
}

@Entity
public class Review {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String rating ;
    private String description;

    @ManyToOne
    private Course course;

}

```

Note : Review is Owning Side of Relationship

```

public void addReviewForCourse() {
    //find a course
    Course course=em.find(Course.class, 1001);

    //creating reviews
    Review review1=new Review("5-star", "Great hand stuff");
    Review review2=new Review("3-star", "Good hand stuff");

    //adding courses to review as Review is woning side of
    relationship
    review1.setCourse(course);
    review2.setCourse(course);

    //Creating ids for reviews to add to DB
    em.persist(review1);
    em.persist(review2);

}
-----ManyToMany-----

```

Note : By Default ManyToMany Relationships are LAZY Fetch \*\*\*

```

create table course (
id int auto_increment primary key,
name varchar(255)
);

create table Student(
id int auto_increment primary key,
name varchar(255)
);

create table student_course (
student_id int ,
course_id int ,
FOREIGN KEY (course_id) REFERENCES course(id),
FOREIGN KEY (student_id) REFERENCES student(id)
);

insert into student values(1001,'Pawan');
insert into student values(1002,'Ranga');
insert into course values(2001,'Java');
insert into course values(2002,'C++');
insert into student_course values(1001,2001);
insert into student_course values(1001,2002);
insert into student_course values(1002,2001);

```

```

@Entity
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    //we have made student to be Owning side of relationship
    @ManyToMany(mappedBy = "courses")
    private List<Student> students=new ArrayList<Student>();

    public List<Student> getStudents() {
        return students;
    }

    public void addStudent(Student student) {
        this.students.add(student);
    }

    public void removeStudent(Student student) {
        this.students.remove(student);
    }
}

@Entity
public class Student {

```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;

private String name;

@ManyToMany
@JoinTable(
    name="student_course",
    joinColumns = @JoinColumn(name="student_id"),
    inverseJoinColumns = @JoinColumn(name="course_id")
)
private List<Course> courses=new ArrayList<>();
}

@Repository
@Transactional
public class StudentRepository {

    @PersistenceContext
    EntityManager em;

    public void fetchStudentAndCourse() {
        Student student=em.find(Student.class, 1001);
        System.out.println(student);
        //Student [id=1001, name=Pawan, courses=[Course [id=2001,
        name=Java], Course [id=2002, name=C++]]]
    }

    public void insertStudentAndCourse() {
        Student student=new Student("Ronit");
        Course course=new Course("BBA");

        // adding student and course
        em.persist(student);
        em.persist(course);

        //owning side is student so it is fine to setup relationship
        from student to course
        student.addCourses(course);
    }
}

```

-----Inheritance With JPA-----

- 1.Inheritance With Single Table
- 2.Inheritance With "Table per Class"
- 3.3.Inheritance with "Joined"

#### 1.Inheritance With Single Table

```

create table Employee_Single_Table(
    Employee_type varchar(255),
    id int auto_increment primary key,
    NAME VARCHAR(255) ,

```

```

        salary int,
        wages_per_hour int
    );

@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "EmployeeType")
class Employee(id, name)

class FullTimeEmployee extends Employee (salary)
class PartTimeEmployee extends Employee (wagesPerHour)

```

## 2. Inheritance With "Table per Class"

Only Table Structure and Inheritance Type changes  
 Change 1: @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)

Change 2:

```

create table Full_Time_Employee(
id int auto_increment primary key,
NAME VARCHAR(255) ,
salary int
);

create table Part_Time_Employee(
id int auto_increment primary key,
NAME VARCHAR(255) ,
wages_per_hour int
);

```

## 3. Inheritance with "Joined"

Change 1: Table Structure`  
 create table Employee(
 id int auto\_increment primary key,
 Name VARCHAR(255)
 );
 create table Full\_Time\_Employee(
 salary int,
 id int,
 FOREIGN KEY (id) REFERENCES Employee(id)
 );

 create table Part\_Time\_Employee(
 wages\_per\_hour int,
 id int ,
 FOREIGN KEY (id) REFERENCES Employee(id)
 );

Change 2 : InheritanceType  
 @Inheritance(strategy = InheritanceType.JOINED)  
-----Inheritance With Single Table-----  
 create table Employee\_Single\_Table(
 Employee\_type varchar(255),
 id int auto\_increment primary key,
 NAME VARCHAR(255) ,
 salary int,
 wages\_per\_hour int
 );

```

@Entity
@Table(name = "Employee_Single_Table")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "EmployeeType")//Column name in table ->
EMPLOYEE_TYPE
public abstract class Employee { // *** This is An Abstract Class

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

}

@Entity
public class FullTimeEmployee extends Employee {

    private int salary;

    @Override
    public String toString() {
        return "FullTimeEmployee [salary=" + salary + ", getId()=" +
getId() + ", getName()=" + getName() + "]";
    }
}

@Entity
public class PartTimeEmployee extends Employee {

    private int wagesPerHour;

    @Override
    public String toString() {
        return "PartTimeEmployee [wagesPerHour=" + wagesPerHour + ",
getId()=" + getId() + ", getName()=" + getName()
+ "]";
    }
}

@Repository
@Transactional
public class EmployeeRepository {

    @PersistenceContext
    EntityManager em;

    //retrieve all employees
    public void getAllEmployees() {
        List<Employee> employees= em.createQuery("select e from Employee
e", Employee.class).getResultList();
        System.out.println(employees);
    }
}

```

```

//insert employees
public void insertEmployee(Employee emp) {
    em.persist(emp);
}
}

@SpringBootApplication
public class HibdemoApplication implements CommandLineRunner{

    private Logger logger=LoggerFactory.getLogger(this.getClass());

    @Autowired
    EmployeeRepository erepo;

    public static void main(String[] args) {
        SpringApplication.run(HibdemoApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        // TODO Auto-generated method stub
        erepo.insertEmployee(new PartTimeEmployee("Kapil", 1000));
        erepo.insertEmployee(new FullTimeEmployee("Pawan", 3000));
        erepo.getAllEmployees();
    }

}

```

Employee_type	id	Name	salary	wagesPerHour
PartTimeEmployee	1	Kapil	Null	1000
FullTimeEmployee	2	Pawan	3000	Null

Note : Employee is Abstract Class here

-----Inheritance With "Table per Class"-----

Only Table Structure and Inheritance Type changes

```

create table Full_Time_Employee(
id int auto_increment primary key,
NAME VARCHAR(255) ,
salary int
);

create table Part_Time_Employee(
id int auto_increment primary key,
NAME VARCHAR(255) ,
wages_per_hour int
);

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)

public abstract class Employee { // *** This is An Abstract Class

```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
private String name;

}

-----Inheritance Shortcut-----
(A) Single Table :
Classes
    Tables
        @Inheritance(strategy=InheritanceType.SINGLE_TABLE) |
        @DiscriminatorColumn(name="emp_type") |
        Abstract Class(id,name)
            |Employee(emp_type,id,name,salary,wagesPerHour)
        class PermanentEmployee(salary) |
        class PartTimeEmployee(wagesPerHour) |

(B) Table Per Class
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS) |
Abstract Class(id,name)
class PermanentEmployee(salary)
    |PermanentEmployee(id,name,salary)
class PartTimeEmployee(wagesPerHour)
    |PartTimeEmployee(id,name,wagesPerHour)

(C) Joined
@Inheritance(strategy=Inheritance.JOINED)
Abstract Class(id,name)
    |Employee(id,name)
class PermanentEmployee(salary)
    |PermanentEmployee(salary,id)
class PartTimeEmployee(wagesPerHour)
    |PartTimeEmployee(wagesPerHour,id)

-----JPQL Queries -----
public void findCourseWithoutStudents() {
    List<Course> courses=em.createQuery("select c from Course c
where c.students is empty", Course.class).getResultList();
    System.out.println(courses);
}

JPQL Query for
1.Course with having atleast 2 students : select c from Course c where
size(c.students) >=2
2.Courses order by size of students : select c from courses order by
size(c.students)
3.Student with passport number like '%1234%' : select s from Student
where s.passport.number like '%1234%'
4.is null
5. between 10 and 1000
6.upper lower trim length

-----JPQL Queries Joins -----
JOIN =>select c,s from Course c JOIN c.students s
LEFT JOIN =>Select c,s from Course c LEFT JOIN c.students s
CROSS JOIN => Select C,s from Course c ,Student s

```

What happens in Cross join : if course has 3 rows and student has 4 rows then we get  $3 \times 4 = 12$  rows in total

```
public void usingJoins() {  
    Query query=em.createQuery("select c,s from Course c JOIN  
c.students s");  
    List<Object[]> resultList=query.getResultList();  
    System.out.println(resultList.size());  
    for(Object[] result:resultList) {  
        System.out.println("Course :" +result[0] + " Student  
:" +result[1]);  
    }  
}
```

-----Generation Strategies in Hiberante/JPA(SITA)-----

1.TABLE  
2.SEQUENCE  
    -Used for Most of databases  
    -@Id  
    @GeneratedValue(  
        strategy = GenerationType.SEQUENCE,  
        generator = "id\_generator"  
    )  
    @SequenceGenerator(  
        name = "id\_generator",  
        sequenceName = "id\_generator\_name",  
        allocationSize = 1  
    )  
    private Integer id;

Note : Once you run your application, you will see a table named `id_generator_name` will get created and initial value will be 1.

logs:  
    Hibernate: create table id\_generator\_name (next\_val bigint)  
engine=MyISAM  
    Hibernate: insert into id\_generator\_name values ( 1 )  
3.IDENTITY  
    -It is not Supported or compatible with Oracle DB  
    - It relies on `auto_increment` database column and let database generate new value on every insertion  
4.Auto  
-For MySql .. Hibernate\_Squence table created in order to maintain `auto_increment` count  
-For Oracle ,PostSql DB ..They have their own sequence mechanism  
-Hiberante most pf times use this startegy

Note : MYSQL supports all startegies  
It depends on Db ki konsa startegies support hoga

-----Transaction Management -DIRTY READ,Non Repeatable Read ,Phantom Read-----

1.Dirty Read -It happens when more than one transaction executes in parallel (together)

                  -Transaction 2 reads the data that has been updated by transaction 1 and transaction 1 is still uncommitted

Example 1: suppose transaction 1 updates a row. Transaction 2 reads the updated row before transaction 1 commits the update

Example 2: Transaction 1->Transfer 50\$ from A to B

Transaction 2->Transfer 100\$ from A to C

A's Account	B's Account	C's Account
200	300	500
Transaction 1 - Step 1   Deduct A's Account by 50\$		150
Transaction 2 - Step 1   Deduct A's Account by 100\$		50
Transaction 1 - Step 2   Deposit B's Account by 50\$		
350		
Transaction 2 - Step 2   Deduct C's Account by 100\$		600

Suppose Transaction 1-Step 2 fails then when have to revert Transaction 1 -Step 1 also ... but transaction 2 (Step-1) has worked on updated value of transaction 1 (Step-1)..so this will bring data inconsistency

Solution : Let one transaction complete first and then the other will start

## 2. Non -Repeatable Read

when a transaction reads the same row twice but gets different data each time

suppose transaction 1 reads a row. Transaction 2 updates or deletes that row and commits the update or delete

Example : Transaction 1 - Step 1 -> select \* from person where id=10  
Person\_Table {id=10 ,name=Ranga ,age=25}

Transaction 2 -Step 1 -> update person set age=30 where  
id=10

Transaction 1 -Step 2 -> select \* from person where id=10  
Person\_Table {id=10 ,name=Ranga ,age=30}

## 3. Phantom Read (har baar number of rows different ho )

when a transaction retrieves a set of rows twice and new rows are inserted into or removed from that set by another transaction that is committed in between.

Example : Transaction 1 - Step 1 -> Select \* from Person where age between 5 and 55

(No of rows  
fetched=3)

Transaction 2 - Step 1 -> insert into person  
values(13,'Ravi', 25)

Transaction 1 - Step 2 ->Select \* from Person where age  
between 5 and 55

(No of rows fetched=4)

-----ISOLATION LEVELS -Read Uncommitted , Read Committed , Repeatable Read , Serialized-----

	Dirty Read	Non repeatable Read	Phantom Read
Read Uncommitted	Not Solved	Not Solved	Not Solved
Read Committed	Solved	Not Solved	Not Solved

Repeatable Read	Solved	Solved	Not Solved
Serialized	Solved	Solved	Solved

Read Committed will lock all Cell Value in DB on Which transaction perfromed.

Repeatable Read will lock row in DB on Which transaction perfromed.  
Serialized will all Rows in DB on Which transaction perfromed and will not allow any insertion.

Table Refered : Person

id	name	age
10	Ranga	20
11	Adam	29
12	Ravi	32

-----Spring vs JPA Transaction Management -----

Note : We have @Transaction that is available in javax.transactionl(JPA) and springframework.transaction(Spring)

1. If there is only one Database to be connected to then we use JPA @Transactional (javax.transactional)

If we have to connect to multiple databases /mq then we use Spring @transactional

2. If we use @Transactional from Spring Framework then we can define isolation level also

@Transactional(isolation=DEFAULT / READ\_COMMITTED/ READ\_UNCOMMITTED /REPEATABLE\_READ /SERIALIZED )

-----CASCADE-----

PERSIST - if the parent entity is persisted then all its related entity will also be persisted.

MERGE - if the parent entity is merged then all its related entity will also be merged.

DETACH - if the parent entity is detached then all its related entity will also be detached.

REFRESH - if the parent entity is refreshed then all its related entity will also be refreshed.

REMOVE - if the parent entity is removed then all its related entity will also be removed.

ALL In this case, all the above cascade operations can be applied to the entities related to parent entity.

-----JPA Entity Lifecycle-----

1.New (Transient)

2.Managed -persist()

3.Detached -detach() - entity is removed from the persistence context

4.Removed -remove() - entity is removed from the database

1.New (Transient): When an entity is first created, it is in the "new" state. At this point, it is not yet associated with any persistence context, and therefore it has no representation in the database.

2.Managed: When an entity becomes associated with a persistence context (usually through the EntityManager.persist() method), it becomes "managed". This means that any changes made to the entity will be persisted to the database when the transaction is committed.

3.Detached: If an entity is removed from the persistence context (usually through the EntityManager.detach() method), it becomes "detached". At this point, it is no longer managed and any changes made to it will not be persisted to the database.

4.Removed: When an entity is removed from the database (usually through the EntityManager.remove() method), it becomes "removed". At this point, it is no longer managed and any changes made to it will not be persisted to the database.

-----Spring data JPA-----

```
CrudRepository (Interface)
|
PagingAndSortingRepository (Interface)
|
JpaRepository (Interface)
```

CrudRepository Methods :

```
findById -> find a course by id
findAll -> find all courses
existsById -> check if course is present
count -> count the number of courses
save -> save the course
delete -> delete the course
```

CrudRepository Methods :

```
flush
saveAndFlush
```

Code For Spring Data JPA :

```
public interface SpringDataCourseRepository extends JpaRepository<Course,
Integer> {
```

```
}
```

```
@SpringBootApplication
public class HibdemoApplication implements CommandLineRunner{
```

```
@Autowired
SpringDataCourseRepository repo;

public static void main(String[] args) {
    SpringApplication.run(HibdemoApplication.class, args);
}

@Override
@Transactional -----> THIS IS IMPORTANT
*****NOTE*****
public void run(String... args) throws Exception {
    //select By Id
    Optional<Course> course =repo.findById(2001);
    if(course.isPresent()) {
        System.out.println(course.get());
    }

    // save and update a course
    Course course1=new Course("Microservices in 100 Steps");
    repo.save(course1);
    course1.setName("Microservices in 100 Steps - Updated");

    //find all Courses
```

```
System.out.println(repo.findAll());  
  
//count all courses  
System.out.println(repo.count());  
  
//Sort Courses by name in desc order  
Sort sort=new Sort(Sort.Direction.DESC,"name");  
//Sort sort=new  
Sort(Sort.Direction.DESC,"name").and(Sort.Direction.DESC,id); --> primary  
and secondary sorting criteria  
System.out.println(repo.findAll(sort));  
  
//Pagination --> suppose u have 100 results and u want to  
divide them in 10 or 20 results per page  
//Page should start from 0 and each page should have 3  
records  
PageRequest pageRequest = PageRequest.of(0, 3);  
Page<Course> firstPage=repo.findAll(pageRequest);  
System.out.println(firstPage.getContent());  
  
Pageable secondPageable=firstPage.nextPageable();  
Page<Course> secondPage=repo.findAll(secondPageable);  
}  
}
```