**Spring Data JPA HandsOn-1**

1. **Explain the need and benefit of ORM.** **ORM Pros and Cons.** **What is ORM?  
   ORM** is a technique in software development that allows you to **interact with a relational database using object-oriented programming languages** (like Java, Python, C#) instead of writing raw SQL queries. An ORM tool (like Hibernate for Java, Entity Framework for C#, SQLAlchemy for Python) **maps database tables to classes**and**rows to objects.**

**Needs:**

Bridges Object-Oriented and Relational Paradigms  
Reduces Boilerplate Code  
Improves Development Speed & Maintainability  
Abstraction from SQL Dialects  
Built-in Features

**Benefits:**Productivity, Maintainability, Portability, Security, Code Reusability

**Cons:**

Performance Overhead, Complex Queries ,Learning Curve, Abstraction Leaks, Less Control

1. **Demonstrate the need and benefit of Spring Data JPA : Evolution of ORM solutions, Hibernate XML Configuration, Hibernate Annotation Configuration, Spring Data JPA, Hibernate benefits, open source, light weight, database independent query**

**Evolution of ORM Solutions**

| Stage | Description |
| --- | --- |
| JDBC | Manual SQL queries, boilerplate code, no object mapping. |
| Hibernate (XML Configuration) | Introduced ORM, XML used for mapping classes to tables. |
| Hibernate (Annotation Configuration) | Used annotations like @Entity, @Id to reduce XML. |

**JDBC (Manual SQL):**

Connection con = DriverManager.getConnection(...);

PreparedStatement stmt = con.prepareStatement("SELECT \* FROM users");

ResultSet rs = stmt.executeQuery();

**Hibernate XML Mapping**:

<class name="User" table="users">

<id name="id" column="id"/>

<property name="name" column="name"/>

</class>

**Hibernate Annotations:**

@Entity

public class User {

@Id

private Long id;

private String name;

}

**Spring Data JPA:**

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByName(String name);

}

**Need for Spring Data JPA**

1. Eliminate Boilerplate Code
   * You don’t have to write save(), findById(), or delete() manually.
2. Rapid Development
   * Focus on business logic; Spring handles the persistence layer.
3. Declarative Queries
   * Auto-generates queries based on method names (findByEmailAndStatus).
4. Built-in CRUD Operations
   * JpaRepository provides all standard operations.
5. Pagination & Sorting
   * Easily paginate and sort with built-in support.
6. Integration with Spring Ecosystem

**3.Explain about core objects of hibernate framewor : Session Factory,**

**Session, Transaction Factory, Transaction, Connection Provider**

**1. Session Factory**

* **What it is:**A thread-safe, heavyweight object created once during application startup.
* **Purpose:**Used to create Session objects to interact with the database.
* **Lifecycle:**  
  Created once per database (typically in the HibernateUtil class) and reused.
* **How it works**:  
  Configured using hibernate.cfg.xml or Java-based configuration.
* **Example:**

SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();

**2. Session**

* **What it is:**A lightweight, non-thread-safe object that represents a connection (unit of work) between the application and the database.
* **Purpose:**  
  Used to perform CRUD operations, create queries, and manage entity lifecycle.
* **Lifecycle:**  
  Created per user request or per transaction. Must be closed after use.
* **Example:** Session session = sessionFactory.openSession();

**3. Transaction Factory**

* **What it is:**An internal Hibernate object responsible for creating Transaction objects.
* **Purpose:**Helps Hibernate choose between JDBC transactions, JTA, or other transaction APIs.
* **Note:**You usually don’t use this directly. It is configured internally.

**4. Transaction**

* **What it is:**Represents a unit of work, typically a single logical operation like saving an object.
* **Purpose:**Ensures atomicity and consistency – changes are either fully committed or rolled back.
* **Example:**

Transaction tx = session.beginTransaction();  tx.commit();

**5. Connection Provider**

* **What it is:**  
  An interface that abstracts how Hibernate gets JDBC connections.
* **Purpose:**  
  Hibernate uses it internally to connect with the database using connection pooling (like HikariCP, C3P0).

**4. Explain ORM implementation with Hibernate XML Configuration and Annotation Configuration**

* + **XML Configuration - persistence class, mapping xml, configuration xml, loading hibernate configuration xml file; Annotation Configuration - persistence class, @Entity, @Table, @Id, @Column, hibernate configuration xml file Loading hibernate configuration and interacting with database get the session factory, open session, begin transaction, commit transaction, close session**

**ORM Implementation with Hibernate XML Configuration**

**1. Persistence Class (POJO)**

public class Student {

private int id;

private String name;

private String email;

}

**2. Mapping XML File (**Student.hbm.xml**)**

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="Student" table="students">

<id name="id" column="id">

<generator class="increment"/>

</id>

<property name="name" column="name"/>

<property name="email" column="email"/>

</class>

</hibernate-mapping>

**3. Hibernate Configuration File (**hibernate.cfg.xml**)**

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/testdb</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">1234</property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>

<property name="hibernate.hbm2ddl.auto">update</property>

<mapping resource="Student.hbm.xml"/>

</session-factory>

</hibernate-configuration>  
**4. Hibernate Main Logic**

import org.hibernate.\*;

public class App {

public static void main(String[] args) {

Configuration cfg = new Configuration();

cfg.configure("hibernate.cfg.xml");

SessionFactory factory = cfg.buildSessionFactory();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Student s = new Student();

s.setName("Alice");

s.setEmail("alice@gmail.com");

session.save(s);

tx.commit();

session.close();

factory.close();

}

}

**ORM Implementation with Hibernate Annotation Configuration  
1. Persistence Class with Annotations**

import jakarta.persistence.\*;

@Entity

@Table(name = "students")

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

@Column(name = "name")

private String name;

@Column(name = "email")

private String email;

}  
**2. Hibernate Configuration File**<hibernate-configuration>

<session-factory>

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/testdb</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">1234</property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>

<property name="hibernate.hbm2ddl.auto">update</property>

<mapping class="Student"/> <!-- class mapping, not XML -->

</session-factory>

</hibernate-configuration>

**3. Hibernate Main Logic (Same as XML)**import org.hibernate.\*;

public class App {

public static void main(String[] args) {

Configuration cfg = new Configuration();

cfg.configure("hibernate.cfg.xml");

SessionFactory factory = cfg.buildSessionFactory();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Student s = new Student();

s.setName("Bob");

s.setEmail("bob@gmail.com");

session.save(s);

tx.commit();

session.close();

factory.close();

}

}

**5.Explain the difference between Java Persistence API, Hibernate and Spring Data JPA**

**JPA (Java Persistence API)**

* It is just a rule book (a specification).
* It tells how to map Java classes to database tables.
* It does not do anything by itself**.**

Example:  
It defines annotations like @Entity, @Id, etc.  
But you need a tool like Hibernate to make it work.

**Hibernate**

* It is a tool (framework) that follows the JPA rules.
* It implements JPA and adds extra powerful features.
* You can use Hibernate with or without JPA.

Example:  
Hibernate can connect to the database, run SQL queries, and manage transactions.

**Spring Data JPA**

* It is a helper tool built on top of JPA + Hibernate.
* It makes your work very easy — you don’t need to write SQL or DAO code.  
    
    
    
  **6.Demonstrate implementation of DML using Spring Data JPA on a single database table Hibernate log configuration and ddl-auto configuration, JpaRepsitory.findById(), defining Query Methods, JpaRespository.save(), JpaRepository.deleteById()   
    
  1. Project Setup Overview**

We'll create a Spring Boot application that performs:

Insert: save()

Read: findById()

Update: save() again

Delete: deleteById()  
  
**2. Entity Class (Persistence Class)**import jakarta.persistence.\*;

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private double price;

}

1. **Repository Interface**

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface ProductRepository extends JpaRepository<Product, Long> {

// Query method

List<Product> findByName(String name); // SELECT \* FROM product WHERE name = ?

}  
  
**4. Service or Main Class to Test DML**import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ProductApp implements CommandLineRunner {

@Autowired

private ProductRepository repo;

public static void main(String[] args) {

SpringApplication.run(ProductApp.class, args);

}

@Override

public void run(String... args) {

// INSERT

Product p1 = new Product();

p1.setName("Laptop");

p1.setPrice(80000);

repo.save(p1);

// SELECT

Product found = repo.findById(p1.getId()).orElse(null);

System.out.println("Found: " + found.getName());

// UPDATE

found.setPrice(75000);

repo.save(found); // save() also updates if ID exists

// DELETE

repo.deleteById(found.getId());

// QUERY METHOD

repo.findByName("Laptop").forEach(p -> System.out.println(p.getId()));

}

}  
  
**5. application.properties Configuration**# Database Connection

spring.datasource.url=jdbc:mysql://localhost:3306/testdb

spring.datasource.username=root

spring.datasource.password=1234

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA & Hibernate

spring.jpa.show-sql=true # Show SQL in logs

spring.jpa.properties.hibernate.format\_sql=true

spring.jpa.hibernate.ddl-auto=update # Auto create/update tables (validate, create, update, none)

# Logging SQL (Hibernate logs)

logging.level.org.hibernate.SQL=DEBUG

logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE