SPRING DATA- JPA–SOLUTIONS:-

**OBJECTIVES**

**1.Explain the need and benefit of ORM.**

**A:-** ORM is a programming technique that lets developers interact with databases using objects, rather than writing SQL queries directly. It maps classes in code to tables in a relational database.

* **Why Do We Need ORM?**

**Abstraction:** ORM abstracts away low-level SQL queries. Developers work with objects, not SQL.  
**Faster Development:** Reduces boilerplate code. No need to manually write INSERTs, UPDATEs, DELETEs.

**Portability:**Makes switching databases easier as ORM handles vendor-specific SQL behind the scenes.  
**Productivity:** Developers can focus on business logic rather than DB query syntax.

* **Benefits of ORM:-**
* **Saves Time** – Less writing, fewer mistakes. ORM does the boring stuff for you.
* **Cleaner Code** – Your code is easier to read and manage.
* **Cross-Database Friendly** – Switch from MySQL to PostgreSQL? No big deal.
* **Safer Transactions** – Handles commits and rollbacks automatically.
* **Handles Relationships** – Like users with posts, orders with products, etc.
* **Drawbacks of ORM:-**
* Performance Overhead (compared to raw SQL for complex queries)
* Learning Curve for complex mappings and behaviors
* Less Control over query optimization and tuning
* Not Ideal for All Scenarios· Demonstrate the need and benefit of Spring Data JPA

**2.Demonstrate the need and benefit of Spring Data JPA.**

**A:-**

* **Evolution of ORM Tools:-**

**Hibernate with XML Configuration (The Old School Way):-**

At first, developers had to write long and clunky XML files to map Java classes to database tables.

It looked something like this:-

**CODE:**-<class name="Book" table="books">

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

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

</class>

**Hibernate with Annotations:-**

Developers got tired of XML, so Hibernate started allowing annotations directly in code:-

@Entity

@Table(name = "books")

public class Book {

@Id

private Long id;

private String title;

}

**Enter Spring Data JPA:-**

Spring Data JPA takes things even further. You just define a Java interface like this:

“public interface BookRepository extends JpaRepository<Book, Long> {}”

Boom. You get **CRUD (Create, Read, Update, Delete)** operations.

**Benefits of Spring Data JPA:-**

**Less Code, More Power**: You focus on business logic; Spring handles the data.  
**Database Independent**: Works with H2, MySQL, PostgreSQL — you name it.  
**Integrates Easily** with Spring Boot (auto-config, dependency injection, etc.)  
**Built on Hibernate**: So you get all the goodness of Hibernate (caching, transactions, performance).  
**Lightweight & Open Source.**

**3. Explain about core objects of hibernate framework**

**A:-**Hibernate is like a helpful middleman between your Java code and your database. To do this job, it uses some important tools, we call them core objects.

**1.** **SessionFactory:-** Think of it as the main engine room of Hibernate.It is created once when the app starts and lives as long as the app runs.It holds all the configuration and metadata for your database connection and mappings.From this factory, we get Sessions (more on that next).

**2.** **Session:-** A single unit of work with the database.You use a session to create, read, update, or delete objects in the database.It's short-lived,usually used per user request or transaction.You get it from the SessionFactory.

**3. TransactionFactory:-** This creates Transaction objects.Hibernate supports working with transactions,so your database operations are safe and consistent.Usually hidden behind Spring or Hibernate’s internals, but it’s there working quietly.

4. **Transaction:-** Represents a single atomic unit of work.Used to make sure operations like save, update, or delete happen fully or not at all.You begin, commit, or rollback a transaction inside a session.

5. **ConnectionProvider:-** A hidden hero that gives Hibernate the actual JDBC connection to the database.Hibernate can use its own built-in provider, or you can plug in custom ones (like HikariCP, C3P0, etc.).

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

**A:- Implement ORM with Hibernate:-**We want to connect a Java class to a database table — that’s what ORM does. Hibernate helps us do that in two main ways:

1. Using XML Configuration
2. Using Annotations in Java code

## **1. XML Configuration Approach:-** This is the **old-school** way. Everything is written in separate XML files.

a. **Persistence Class** (Java POJO):-

CODE:- public class Student {

private int id;

private String name;

private String email;

}

### **b. Mapping XML File (Student.hbm.xml):-**This file tells Hibernate: “Here’s how this class maps to a table

CODE:- <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>

### **c. Configuration File (hibernate.cfg.xml):-**This is where you set up your DB connection and mapping file.

<hibernate-configuration>

<session-factory>

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

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

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

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

<!-- Mapping -->

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

</session-factory>

</hibernate-configuration>

d. **Steps to Interact with DB:-**

Configuration cfg = new Configuration();

cfg.configure("hibernate.cfg.xml"); // Load config

SessionFactory factory = cfg.buildSessionFactory();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Student student = new Student();

student.setName("Spandana");

student.setEmail("spandana@email.com");

session.save(student);

tx.commit(); // Save to DB

session.close();

## **2. Annotation Configuration Approach (Modern & Cleaner Way)**

No need for XML mapping. You add annotations **directly in the Java class**.

a. **Persistence Class with Annotations:-**

import javax.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;

}

**b. Hibernate Configuration XML:-**

<hibernate-configuration>

<session-factory>

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

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

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

<!-- Add annotated class -->

<mapping class="com.example.Student"/>

</session-factory>

</hibernate-configuration>

c. **Interacting with DB:-**

Configuration cfg = new Configuration();

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

SessionFactory factory = cfg.buildSessionFactory();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Student student = new Student();

student.setName("Spandana");

student.setEmail("spandana@email.com");

session.save(student);

tx.commit();

session.close();

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

**A:- 1. JPA (Java Persistence API):-**JPA is just a specification, a set of guidelines (defined in JSR 338).It tells how Java objects should be mapped to a database but doesn't do the actual work.You can't use JPA directly. You need a tool that implements JPA to make it work.

**2. Hibernate:-**Hibernate is a popular ORM framework that implements JPA.You can use it with or without JPA.Hibernate also adds extra features like caching, dirty checking, lazy loading, etc.

### **3. Spring Data JPA:-** Spring Data JPA is a Spring module that sits on top of JPA and Hibernate.It removes boilerplate code like save(), findById(), and findByName() , just use method names and Spring generates the queries.Integrates easily with Spring Boot.

**What is JPA?**

JPA is **just a specification** , think of it like a *set of guidelines* for storing Java objects in a database.

It defines how to:

* Map Java classes to database tables
* Handle relationships (like OneToMany, ManyToOne)
* Manage transactions and queries

**JPA itself doesn’t do anything** — it needs a real ORM tool like Hibernate to bring it to life.

**What is Hibernate?**

Hibernate is a **real ORM tool** that actually **implements** JPA.It does the heavy lifting: mapping, saving, querying, managing database interactions.You can use it with or without JPA.

It has extra goodies like:

* Caching
* Lazy/eager loading
* HQL (Hibernate Query Language)

What is Spring Data JPA?

Spring Data JPA is part of the **Spring ecosystem**.It sits on top of JPA and Hibernate and **makes things ridiculously easier**.

Just define interfaces like:-

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByName(String name);

}

6. **Demonstrate implementation of DML using Spring Data JPA on a single database table.**

**A:-** Goal: Perform DML Using Spring Data JPA

Let's say we have a simple table: students (id, name, email)

We'll:

* Create a student
* Fetch a student by ID
* Update student info
* Delete a student

1. **application.properties Configuration:-**

spring.datasource.url=jdbc:mysql:

spring.datasource.username=root

spring.datasource.password=yourpassword

# JPA + Hibernate settings

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

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

2. **Entity Class: Student.java:-**

import jakarta.persistence.\*;

@Entity

@Table(name = "students")

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

}

3. **Repository Interface:-**

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

public interface StudentRepository extends JpaRepository<Student, Long> {

// Custom Query Method (auto-generated)

List<Student> findByName(String name);

}

4. **Service or Runner Class to Do DML:-**

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

@Component

public class StudentTestRunner implements CommandLineRunner {

@Autowired

private StudentRepository repo;

@Override

public void run(String... args) {

Student s1 = new Student();

s1.setName("Spandana");

s1.setEmail("spandana@email.com");

repo.save(s1);

repo.findById(1L).ifPresent(student -> {

System.out.println("Fetched: " + student.getName());

});

repo.findByName("Spandana").forEach(s -> {

System.out.println("Found: " + s.getEmail());

});

repo.findById(1L).ifPresent(student -> {

student.setEmail("newemail@abc.com");

repo.save(student); // this acts like update

});

repo.deleteById(1L);

}

}

### **Output (In Console):-**

With show-sql=true, Hibernate will print something like:

insert into students (email, name) values (?, ?)

select \* from students where id=?

select \* from students where name=?

update students set email=? where id=?

delete from students where id=?

**7. Demonstrate implementation of Query Methods feature of Spring Data JPA**

Spring Data JPA gives you **smart method names** like findByNameContaining, and it automatically generates the SQL for you.

**Setup: Our Entity:-**

@Entity

@Table(name = "students")

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

private int age;

private LocalDate joinedDate;

}

**Repository Interface:-**

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

import java.time.LocalDate;

import java.util.List;

public interface StudentRepository extends JpaRepository<Student, Long> {

List<Student> findByNameContaining(String keyword);

List<Student> findByEmailStartingWith(String prefix);

List<Student> findByOrderByNameAsc(); // or Desc

List<Student> findByJoinedDateBetween(LocalDate start, LocalDate end);

List<Student> findByAgeGreaterThan(int age);

List<Student> findByAgeLessThan(int age);

List<Student> findTop3ByOrderByAgeDesc(); }

**Using These in Code:-**

@Component

public class StudentQueryDemo implements CommandLineRunner {

@Autowired

StudentRepository repo;

@Override

public void run(String... args) {

repo.findByNameContaining("pan").forEach(s -> System.out.println("Found: " + s.getName()));

repo.findByEmailStartingWith("span").forEach(s -> System.out.println("StartsWith: " + s.getEmail()));

repo.findByOrderByNameAsc().forEach(s -> System.out.println("Sorted: " + s.getName()));

LocalDate start = LocalDate.of(2024, 1, 1);

LocalDate end = LocalDate.of(2025, 1, 1);

repo.findByJoinedDateBetween(start, end).forEach(s -> System.out.println("Between: " + s.getName()));

repo.findByAgeGreaterThan(18).forEach(s -> System.out.println("Adult: " + s.getName()));

repo.findByAgeLessThan(18).forEach(s -> System.out.println("Minor: " + s.getName()));

repo.findTop3ByOrderByAgeDesc().forEach(s -> System.out.println("Top Age: " + s.getAge()));

}

}

8. **Demonstrate implementation of O/R Mapping**

A;- When using JPA with databases, we often need to model **relationships between tables**. In code, this means defining relationships between **Java classes (entities)**.

We’ll explore:

* @ManyToOne + @JoinColumn
* @OneToMany
* FetchType.LAZY vs FetchType.EAGER
* @ManyToMany + @JoinTable
* mappedBy for bidirectional mapping

Imagine you have:

* A Student can belong to **one** Department → @ManyToOne
* A Department can have **many** Students → @OneToMany
* A Student can enroll in **many** Courses, and a Course can have **many** Students → @ManyToMany

**1. @ManyToOne and @OneToMany:-**

@Entity

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "department", fetch = FetchType.LAZY)

private List<Student> students = new ArrayList<>();

}

**Entity: Student.java:-**

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private long id;

private String name;

@ManyToOne(fetch = FetchType.EAGER)

@JoinColumn(name = "department\_id") // FK column in student table

private Department department;

}

**Fetch Types:-**

| **EAGER:-** Loads the associated object immediately with the main object  **LAZY:-** Loads the associated object **only when you access it (on-demand)** |
| --- |

| **2. @ManyToMany and @JoinTable:-**  @Entity  public class Course {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String title;  @ManyToMany(mappedBy = "courses")  private List<Student> students = new ArrayList<>();  }  **Update Entity: Student.java**  @ManyToMany  @JoinTable(  name = "student\_course",  joinColumns = @JoinColumn(name = "student\_id"),  inverseJoinColumns = @JoinColumn(name = "course\_id")  )  private List<Course> courses = new ArrayList<>();  **9. Demonstrate writing Hibernate QueryLanguage**  **and Native Query** |
| --- |

A:-HQL (Hibernate Query Language) and JPQL (Java Persistence Query Language) are both object-oriented query languages used to interact with data in JPA-based applications. They are almost identical in syntax and purpose, with JPQL being the official specification under JPA and HQL being Hibernate’s specific implementation. Both use **entity names** and **Java class field names** instead of table or column names, which makes them database-independent and easier to maintain when the schema changes. They support relationships between entities and can use features like JOIN FETCH for eager loading. On the other hand, **Native Queries** are plain SQL statements written directly against the database tables and columns, just like in traditional JDBC. They are defined using the @Query annotation with nativeQuery = true. Native queries provide full control and may offer better performance for complex or DB-specific operations, but they are **not portable** across different databases and bypass JPA’s abstraction layer. So, while JPQL/HQL is safer and more maintainable for most cases, native SQL is powerful when you need fine-grained performance or features specific to your database engine.

**10. Explain the need and benefit of Criteria Query.**

A:-In most JPA applications, we often use **JPQL (Java Persistence Query Language)** or **@Query** annotations to fetch data. But in some situations, writing queries as strings becomes **messy, less readable, or even error-prone**.

**Why Use Criteria Query?**

**Dynamic Query Building:-**When your query conditions **depend on user input**, you can dynamically add filters using Criteria.

* For example: filter by name if provided, by age if selected, and by status if checked.

**Type-Safe Query Writing**

* With JPQL strings, if you mistype a field name, it’ll only fail at runtime.
* But Criteria API uses **Java class references**, so mistakes are caught **at compile time**.

**Avoid String Concatenation Mess**

* With dynamic filtering in JPQL, you often end up writing long concatenated query strings.
* Criteria API keeps everything neat and modular — like building a query Lego-style

**Example: Fetch students older than 18:-**

@Autowired

private EntityManager em;

public List<Student> getAdults() {

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Student> cq = cb.createQuery(Student.class);

Root<Student> root = cq.from(Student.class);

cq.select(root).where(cb.greaterThan(root.get("age"), 18));

TypedQuery<Student> query = em.createQuery(cq);

return query.getResultList();

}