1Q.

What is ORM (Object-Relational Mapping)?

ORM is a programming technique that allows developers to **interact with a database using the programming language's object-oriented paradigm** instead of writing raw SQL queries.

ORM tools map database **tables** to **classes**, and **rows** to **objects**. For example, a User table becomes a User class, and each user record becomes a User object.

Need for ORM

1. **Bridges the gap between object-oriented code and relational databases**.
2. Avoids writing repetitive SQL for CRUD (Create, Read, Update, Delete) operations.
3. Reduces boilerplate code, making development **faster and cleaner**.
4. Abstracts database logic – developers don't need deep SQL knowledge.
5. Makes **database switching** easier (e.g., switching from MySQL to PostgreSQL).

Benefits of ORM

1. **Faster Development**: Saves time writing SQL queries.
2. **Cleaner Code**: Interact with DB through objects, not raw queries.
3. **Improved Security**: Prevents SQL injection by using parameterized queries.
4. **Portability**: Code works across different DB systems with minimal changes.
5. **Maintainability**: Easier to understand and maintain due to consistent structure.

ORM Pros and Cons

Pros:

- Rapid application development
- Reduced risk of SQL injection
- Easier to manage relationships (1:1, 1:N, N:M)
- Built-in migration tools in many ORMs

Cons:

- Slower than raw SQL in complex queries
- Learning curve for beginners
- Less control over generated SQL (which can hurt performance)
- Can become heavy for simple applications

Examples of ORM Tools:

- **Java** – Hibernate
- **Python** – SQLAlchemy, Django ORM
- **JavaScript/Node.js** – Sequelize, TypeORM
- **C#** – Entity Framework

2Q.

Demonstrate the Need and Benefit of Spring Data JPA

Evolution of ORM Solutions:

Stage	Description
1. JDBC (Manual SQL)	Developers wrote raw SQL using JDBC APIs. This was verbose, error-prone, and difficult to maintain.
2. Hibernate with XML Configuration	Introduced ORM — allowed Java classes to be mapped to database tables via XML config. Reduced manual SQL, but managing XML files was cumbersome.
3. Hibernate with Annotations	Replaced XML config with Java annotations like @Entity, @Table, and @Column. Easier and more readable.
4. Spring + Hibernate	Spring provided transaction management and DAO abstraction. Still required developers to write DAO code and manage sessions.
5. Spring Data JPA	Further abstraction layer over JPA (Hibernate). Eliminates boilerplate code — CRUD operations and queries are derived from method names.

Why Spring Data JPA?

Spring Data JPA is part of the larger Spring Data family. It simplifies JPA-based repositories and reduces boilerplate code, offering:

Key Benefits:

- **Faster development** – no need to write implementation for common queries.
- **Clean code** – uses method naming conventions to generate queries automatically.
- **Built-in pagination and sorting**.
- **Safe queries** – prevents SQL injection through parameterized queries.
- **Easily switch between databases** (e.g., H2 to MySQL) without major code changes.
- **Full integration with Spring Boot** – easy to set up with auto-configuration.

Hibernate Benefits (Under the Hood of Spring Data JPA)

- Open Source & Widely Adopted
- Supports Lazy/Eager Fetching
- Automatic Table Creation
- Powerful Query Language (HQL)
- Database Independence

3Q.

Core Objects of Hibernate Framework

Hibernate is an ORM (Object-Relational Mapping) framework that helps Java applications interact with databases easily and efficiently. Below are the **key components (core objects)** that make Hibernate work:

SessionFactory

- **What is it?**
  A thread-safe, heavyweight object created once per application and used to obtain Session objects.
- **Role:**
  - Used to configure Hibernate (using hibernate.cfg.xml)
  - Maintains second-level cache (optional)
  - Responsible for creating Session instances
- **Created using:**
  Configuration().configure().buildSessionFactory();
- **Lifecycle:**
  Created once and reused across the entire app.

Session

- **What is it?**
  A lightweight, non-thread-safe object used for performing CRUD operations.
- **Role:**
  - Represents a single unit of work with the database
  - Provides methods like save(), update(), get(), delete(), createQuery()
  - Manages the first-level (session) cache

- **Lifecycle:**
  Short-lived, created per request or per transaction.

TransactionFactory

- **What is it?**
  Factory that helps create Transaction instances.
- **Role:**
  Abstracts the creation of Transaction objects based on the underlying transaction management system.
- **Note:**
  You rarely interact with TransactionFactory directly unless you're writing custom transaction strategies.

Transaction

- **What is it?**
  Represents a single atomic unit of work (e.g., multiple DB operations) that can be committed or rolled back.
- **Role:**
  - Begins and ends a transaction using beginTransaction(), commit(), or rollback().
  - Ensures data integrity and consistency
- **Typical usage:**
- Transaction tx = session.beginTransaction();
- // perform DB operations
- tx.commit(); // or tx.rollback();

ConnectionProvider

- **What is it?**
  Interface used by Hibernate to get JDBC connections from the database.
- **Role:**
  - Abstracts the connection details
  - Works behind the scenes to connect Hibernate with your actual database
- **Types:**
  Hibernate provides built-in connection providers, or you can define your own (e.g., for connection pooling).

4Q.

1. ORM Implementation Using Hibernate XML Configuration

Components:

a. Persistence Class

A normal Java POJO (Plain Old Java Object) representing a table.

```
public class Employee {
   private int id;
   private String name;
   private double salary;

   // Getters and Setters
}
```

b. Mapping XML File (Employee.hbm.xml)

Maps the class and its properties to the database table and columns.

```
<hibernate-mapping>
   <class name="Employee" table="employee">
      <id name="id" column="id">
         <generator class="increment"/>
      </id>
      <property name="name" column="name"/>
      <property name="salary" column="salary"/>
   </class>
</hibernate-mapping>
```

c. Hibernate Configuration XML (hibernate.cfg.xml)

Defines database connection and includes mapping files.

```
<hibernate-configuration>
  <session-factory>
   <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
   <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">password</property>

    <!-- Mapping File -->
    <mapping resource="Employee.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Steps to Interact with DB:

```
Configuration cfg = new Configuration().configure();       // Load cfg file
SessionFactory factory = cfg.buildSessionFactory();        // Get SessionFactory
Session session = factory.openSession();                   // Open Session
Transaction tx = session.beginTransaction();               // Begin Transaction

Employee e = new Employee();
e.setName("Rekha");
e.setSalary(50000);
session.save(e);                                           // DB Operation

tx.commit();                                               // Commit
session.close();                                           // Close Session
```

2. ORM Implementation Using Hibernate Annotation Configuration

Components:

a. Persistence Class

Uses annotations instead of XML mapping.

```
import jakarta.persistence.*;

@Entity
@Table(name = "employee")
public class Employee {

    @Id
```

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

    @Column(name = "name")
    private String name;

    @Column(name = "salary")
    private double salary;

    // Getters and Setters
}
```

b. Hibernate Configuration XML (hibernate.cfg.xml)

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <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">password</property>

    <!-- Class Mapping -->
    <mapping class="Employee"/>
  </session-factory>
</hibernate-configuration>
```

Steps to Interact with DB (Same as XML):

```
Configuration cfg = new Configuration().configure();       // Load cfg file
SessionFactory factory = cfg.buildSessionFactory();        // Get SessionFactory
Session session = factory.openSession();                   // Open Session
Transaction tx = session.beginTransaction();               // Begin Transaction

Employee e = new Employee();
```

```
e.setName("Rekha");
e.setSalary(70000);
session.save(e);                    // Save object

tx.commit();                        // Commit
session.close();                    // Close session
```

Comparison

Feature	**XML Configuration**	**Annotation Configuration**
Mapping Style	Separate .hbm.xml files	Inline with Java code using annotations
Readability	Less readable	More readable
Maintainability	Separate mapping	Easier with annotations
Standard Practice	Older	Modern and widely used

5.

Term	**Type**	**Role**
JPA	Specification	Defines standard rules for ORM in Java (JSR 338)
Hibernate	Implementation	A popular ORM tool that implements JPA
Spring Data JPA	Abstraction Layer	Built on top of JPA/Hibernate to eliminate boilerplate

1. Java Persistence API (JPA)

What is JPA?

- **JPA** is **only a specification**, not a framework or library.
- Defined by **JSR 338** (Java Specification Request).
- It provides **standard interfaces and annotations** to manage relational data in Java.

Key Features:

- Annotations like @Entity, @Id, @Table, @OneToMany, etc.

- Entity lifecycle management
- JPQL (Java Persistence Query Language)

2. Hibernate

What is Hibernate?

- Hibernate is a **framework/tool** and a **JPA implementation**.
- Provides ORM functionality — maps Java objects to database tables.
- Supports both:
    - **Native Hibernate API** (e.g., Session, Criteria, HQL)
    - **JPA API** (if configured to use JPA)

Hibernate Extra Features (beyond JPA):

- First and second-level caching
- Custom types and interceptors
- Advanced performance tuning
- Support for native SQL and HQL (Hibernate Query Language)

3. Spring Data JPA

What is Spring Data JPA?

- Part of the **Spring Data** project.
- It is an **abstraction layer** over JPA (usually backed by Hibernate).
- Removes **boilerplate code** like DAO implementations.
- Uses method name conventions to auto-generate queries.

Features:

- Auto-implemented interfaces like JpaRepository, CrudRepository
- Built-in pagination, sorting
- Custom query support using JPQL or native SQL
- Easy integration with **Spring Boot**

Example:

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
   List<Employee> findByName(String name); // No implementation needed!
}
```

6.

Perform DML Operations on a Single Table (e.g., Student)

Technologies:

- Spring Boot
- Spring Data JPA
- H2 (in-memory) or MySQL
- Hibernate (under-the-hood)

Project Setup (pom.xml)

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId> <!-- or use MySQL -->
    <artifactId>h2</artifactId>
  </dependency>
</dependencies>
```

Entity Class: Student.java

```
import jakarta.persistence.*;

@Entity
public class Student {

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

```
    private String name;
    private String department;
    private int marks;

    // Getters and Setters
}
```

Repository Interface: StudentRepository.java

```
import org.springframework.data.jpa.repository.JpaRepository;
import java.util.List;

public interface StudentRepository extends JpaRepository<Student, Long> {

    // Query method: Find by department
    List<Student> findByDepartment(String department);

    // Query method: Find students with marks greater than a value
    List<Student> findByMarksGreaterThan(int marks);
}
```

Service or Runner to Demonstrate DML: StudentService.java

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
import java.util.Optional;

@Component
public class StudentService implements CommandLineRunner {

    private final StudentRepository repo;

    public StudentService(StudentRepository repo) {
```

```
    this.repo = repo;
}

@Override
public void run(String... args) {

    // INSERT
    Student s1 = new Student();
    s1.setName("Rekha");
    s1.setDepartment("CSE");
    s1.setMarks(90);
    repo.save(s1); // save() inserts

    // READ
    Optional<Student> found = repo.findById(1L);
    found.ifPresent(System.out::println);

    // UPDATE
    if (found.isPresent()) {
        Student s = found.get();
        s.setMarks(95);
        repo.save(s); // save() updates
    }

    // DELETE
    repo.deleteById(1L);

    // QUERY METHOD
    repo.findByDepartment("CSE").forEach(System.out::println);
    repo.findByMarksGreaterThan(70).forEach(System.out::println);
}
```

}

Configuration: application.properties

```
# Database config
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# Hibernate config
spring.jpa.show-sql=true                    # Log SQL queries
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.hibernate.ddl-auto=update        # auto create/update table
logging.level.org.hibernate.SQL=DEBUG       # Hibernate SQL logs
logging.level.org.hibernate.type.descriptor.sql=TRACE
```

ddl-auto options:

- none – No schema generation
- update – Auto-updates schema based on entities (ideal for dev)
- create – Re-creates schema on every run
- create-drop – Drops schema after app shutdown

Output Example:

```
Hibernate: insert into student (department, marks, name) values (?, ?, ?)
Student{id=1, name='Rekha', department='CSE', marks=90}
Hibernate: update student set marks=? where id=?
Hibernate: delete from student where id=?
```