Karthika Vijay - 6377005

OBJECTIVES

- **Explain the need and benefit of ORM**
  - **ORM Pros and Cons**

    Pros:

    - Simplifies database interactions.
    - Saves time and effort for common operations.
    - Integrates well with object-oriented design.
    - Reduces chances of SQL errors and vulnerabilities.

    Cons:

    - Can produce inefficient SQL queries (especially for complex joins).
    - May introduce overhead and reduce performance in large-scale applications.
    - Learning curve for understanding ORM behavior and lifecycle.
    - Less control over fine-tuned database operations.

  - **What is ORM?**

    Object-Relational Mapping (ORM) is a programming technique that allows developers to interact with a relational database using the object-oriented paradigm. ORM frameworks map database tables to classes, rows to objects, and columns to attributes.

- **Demonstrate the need and benefit of Spring Data JPA**
  - **With H2 In-Memory Database**

    - Useful for quick testing and prototyping.
    - No installation needed.
    - In-memory, so data is lost on restart.

```
@Entity
public class Customer {
   @Id @GeneratedValue
   private Long id;
```

```
    private String name;
}

public interface CustomerRepository extends JpaRepository<Customer, Long>
{
    List<Customer> findByName(String name);
}
```

- **With MySQL**
  - Suitable for real-world production systems.
  - Uses the same code as H2, just with different database configuration.

Employee.java

```
@Entity
public class Employee {
    @Id @GeneratedValue
    private Long id;
    private String name;
}
```

EmployeeRepository.java

```
public interface EmployeeRepository extends JpaRepository<Employee,
Long> {
    List<Employee> findByName(String name);  // auto-generates query!
}
```

EmployeeController.java

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {
    @Autowired
    private EmployeeRepository repo;

    @GetMapping
    public List<Employee> getAll() {
        return repo.findAll();
    }
}
```

- **XML Configuration**
  - Mapping and configuration done using *.xml* files.
  - Requires manual setup of *hibernate.cfg.xml* and mapping files like *employee.hbm.xml*.

Hibernate.cfg.xml

```
<!-- hibernate.cfg.xml -->
<hibernate-configuration>
  <session-factory>
    <property
name="connection.url">jdbc:mysql://localhost:3306/test</property>
    <property name="connection.username">root</property>
    <mapping resource="employee.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Employee.hbm.xml

```
<!-- employee.hbm.xml -->
<class name="Employee" table="EMPLOYEE">
  <id name="id" column="ID"/>
  <property name="name" column="NAME"/>
```

```
</class>
```

- **Hibernate Configuration**
  - Uses Java annotations (*@Entity, @Table, @Id,* etc.) instead of external XML files.
  - Still uses *SessionFactory* manually for DB operations.

```
@Entity
@Table(name="EMPLOYEE")
public class Employee {
   @Id
   private int id;


   private String name;
}


SessionFactory factory = new
Configuration().configure().buildSessionFactory();
Session session = factory.openSession();
Transaction tx = session.beginTransaction();
session.save(new Employee(1, "John"));
tx.commit();
```

- **Explain the difference between Java persistence API, Hibernate and Spring data JPA**
  - **Java persistence API (JPA)**
    - JPA is a specification (defined in JSR 338).
    - It defines standard APIs and annotations for object-relational mapping (ORM) in Java.
    - JPA itself does not provide any implementation — it's just a contract (like an interface).
  - **Hibernate**
    - Hibernate is an implementation of the JPA specification.
    - It’s also an independent ORM tool with extra features beyond JPA.
  - **Spring Data JPA**

- Spring Data JPA is a Spring module built on top of JPA and Hibernate.
- It provides a higher level abstraction to:
  - Automatically create repositories
  - Reduce boilerplate code
  - Auto-generate queries using method names