**Topic-Spring Data JPA with Spring Boot, Hibernate**

**Q) ORM Learn — Spring Data JPA Quick Example (Oracle Version)**

**pom.xml (Adding Dependencies)**I added the following dependencies to my Maven project so that Spring Boot knows it has to use Spring Data JPA and Oracle DB.

**pom.xml:**

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.oracle.database.jdbc</groupId>

<artifactId>ojdbc8</artifactId>

<version>19.3.0.0</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

**application.properties**

application.properties:

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe

spring.datasource.username=system

spring.datasource.password=oracle

spring.datasource.driver-class-name=oracle.jdbc.OracleDriver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.database-platform=org.hibernate.dialect.Oracle12cDialect

Entity Class — Student.java  
Student.java:

java

import jakarta.persistence.\*;

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String course;

}

Repository Interface — StudentRepository.java  
Instead of writing queries manually, I just created an interface that extends JpaRepository. Spring automatically gives me all CRUD methods like save(), findAll(), deleteById(), etc.

**StudentRepository.java:**

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

public interface StudentRepository extends JpaRepository<Student, Long> {

}

**REST Controller — StudentController.java**

This is where I exposed endpoints like POST, GET, DELETE to interact with the database.

**StudentController.java:**

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

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/students")

public class StudentController {

@Autowired

private StudentRepository studentRepository;

@PostMapping

public Student addStudent(@RequestBody Student student) {

return studentRepository.save(student);

}

@GetMapping

public List<Student> getAllStudents() {

return studentRepository.findAll();

}

@DeleteMapping("/{id}")

public void deleteStudent(@PathVariable Long id) {

studentRepository.deleteById(id);

}

}

Main Application — Application.java  
My main class to start the Spring Boot application.

**Application.java:**

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

**Q) Difference between JPA, Hibernate and Spring Data JPA**

JPA (Java Persistence API) is just a specification provided by Java to manage relational data in Java applications. Think of it as a set of guidelines or rules on how to persist (save and retrieve) data. It doesn’t do anything by itself. It needs an implementation (like a car blueprint that needs a manufacturer to build it).

Hibernate is one of the most popular implementations of the JPA specification. It follows the rules set by JPA but also adds many extra features on top of it, making developers' lives easier.

Spring Data JPA is a part of the Spring ecosystem. It builds on top of JPA and Hibernate to make working with databases even simpler. It reduces the amount of code you need to write. For example, you can create a repository interface and Spring will automatically generate the implementation for basic CRUD operations without writing a single query.

**🔹 Java Persistence API (JPA)**

* JPA is a **Java specification (JSR 338)** for persisting, reading, and managing data between Java objects and relational databases.
* It provides a set of **interfaces and annotations**, but does **not include any implementation**.
* JPA defines the standard for ORM, and frameworks like Hibernate act as implementations of this specification.

**🔹 Hibernate**

* Hibernate is an **Object-Relational Mapping (ORM) tool** that implements the JPA specification.
* It provides additional features beyond the JPA spec like caching, native queries, etc.
* Requires manual handling of sessions and transactions.

**🔹 Spring Data JPA**

* Spring Data JPA is a **Spring module** that provides an **abstraction layer over JPA**.
* It does **not implement JPA**, but depends on providers like Hibernate underneath.
* It significantly reduces **boilerplate code** and integrates transaction management.
* Supports **method-based query generation**, pagination, and more.

**Key Differences**

| **Feature** | **JPA** | **Hibernate** | **Spring Data JPA** |
| --- | --- | --- | --- |
| **Type** | Specification | ORM Framework / JPA Implementation | Abstraction Layer over JPA |
| **Scope** | Standard API for ORM | Full-fledged ORM solution | Simplifies JPA usage in Spring |
| **Dependency** | Requires an implementation | Can work standalone or as JPA provider | Requires JPA and an ORM (e.g., Hibernate) |
| **API** | EntityManager, JPQL | HQL, Session API, JPA APIs | Repository interfaces, Query methods |
| **Use Case** | Portable ORM code | Feature-rich ORM | Simplified data access in Spring |
| **Learning Curve** | Moderate | Steeper due to extra features | Easiest, but requires Spring knowledge |
| **Flexibility** | Limited to spec | Highly customizable | Abstracts complexity, less flexible |

**How They Work Together**

* **JPA** provides the standard interface and annotations.
* **Hibernate** implements JPA and adds its own advanced features.
* **Spring Data JPA** builds on JPA (often using Hibernate as the provider) to simplify data access in Spring applications.
* In a typical Spring Boot application:
  + You use Spring Data JPA repositories for data access.
  + Spring Data JPA delegates to Hibernate for ORM functionality.
  + Hibernate implements the JPA specification to interact with the database.

**🔸 Code Comparison**

**🔸 Hibernate Example**

public Integer addEmployee(Employee employee) {

Session session = factory.openSession();

Transaction tx = null;

Integer employeeID = null;

try {

tx = session.beginTransaction();

employeeID = (Integer) session.save(employee);

tx.commit();

} catch (HibernateException e) {

if (tx != null) tx.rollback();

e.printStackTrace();

} finally {

session.close();

}

return employeeID;

}

**🔸 Spring Data JPA Example**

**🔹 EmployeeRepository.java**

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

**🔹 EmployeeService.java**

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

employeeRepository.save(employee);

}

**Behind the Scenes**

* **JPA:** Defines the annotations and contract for persistence (e.g., @Entity, @Id).
* **Hibernate:** Implements the JPA contract and executes DB operations like INSERT, SELECT.
* **Spring Data JPA:**
* Automatically generates the repository code (e.g., save, findAll).
* Internally uses Hibernate as the JPA provider (unless configured otherwise).
* Handles session & transaction lifecycle so you don’t have to.

**Conclusion**

* Use **JPA** for portable, standard ORM code.
* Use **Hibernate** when you need advanced ORM features or are not using Spring.
* Use **Spring Data JPA** for simplified data access in Spring applications, leveraging Hibernate as the JPA provider.