Data Access with Spring Boot (Spring Data JPA)

Enterprise applications are heavily data-driven. They involve creating, reading, updating, and deleting information from databases in a structured and efficient way. Traditionally, developers had to write long SQL statements and boilerplate JDBC code to handle these operations. Spring Boot, with the help of **Spring Data JPA**, simplifies this process drastically by providing a powerful abstraction over the database layer.

In this section, we will understand the **core concepts of Spring Data JPA**, learn how to configure databases, create entity classes, and implement repositories. We will also explore advanced concepts such as JPQL queries, ORM principles, and the use of H2 and MySQL databases in real-world scenarios.

1. Understanding Spring Data JPA

JPA (Java Persistence API) is a specification that defines how Java objects can be mapped to relational database tables. The most popular implementation of JPA is **Hibernate**, which is widely used in enterprise applications.

Spring Data JPA builds on top of JPA and Hibernate and reduces the boilerplate code required for database operations. Instead of writing `INSERT`, `UPDATE`, and `SELECT` statements manually, developers can rely on predefined methods and conventions provided by Spring Data JPA.

For example, to retrieve all student records from a database, you don't need to write SQL. A single method call like `findAll()` is enough.

💡 **Interview Insight:** If asked in an interview why Spring Data JPA is preferred, highlight that it provides **declarative repositories, automatic query generation, support for pagination, and integration with Spring Boot's ecosystem**.

2. Configuring Databases in Spring Boot

Spring Boot makes database configuration extremely simple through the `application.properties` or `application.yml` file.

Example: Configuring MySQL

```
spring.datasource.url=jdbc:mysql://localhost:3306/student_db
spring.datasource.username=root
spring.datasource.password=admin123
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

- `spring.datasource.url` specifies the connection string.
- `spring.jpa.hibernate.ddl-auto=update` ensures the schema is updated automatically.
- `spring.jpa.show-sql=true` helps developers by printing executed queries in the console.

Example: Configuring H2 (In-Memory Database)

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
```

The H2 database is commonly used for **unit testing and demos** because it runs in memory and does not require installation. In production, however, persistent databases like MySQL or PostgreSQL are used.

💡 **Real-World Example:** During development, a team might use H2 for quick testing and switch to MySQL in staging and production environments without changing application code. Only configuration changes are needed.

3. Creating Entity Classes

An entity class represents a database table. Each object of the entity corresponds to a row in the table.

```
import jakarta.persistence.*;

@Entity
@Table(name = "students")
public class Student {

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

    @Column(nullable = false)
    private String name;

    @Column(unique = true, nullable = false)
    private String email;

    private int age;

    // Getters and Setters
}
```

- `@Entity` marks the class as a database entity.
- `@Id` and `@GeneratedValue` define the primary key.
- `@Column` allows customization such as uniqueness or non-null constraints.

💡 **Use Case:** In a *Student Management System*, each student's record is mapped to the `students` table in the database. The ORM framework automatically generates the necessary `INSERT` or `SELECT` statements.

4. Repository Interfaces

Instead of writing SQL queries manually, developers can define repository interfaces. These repositories extend `JpaRepository`, which provides ready-made methods such as `save()`, `findById()`, `deleteById()`, and `findAll()`.

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
   Student findByEmail(String email);
   List<Student> findByAgeGreaterThan(int age);
}
```

Here:

- `findByEmail()` automatically generates a query to fetch a student by email.
- `findByAgeGreaterThan(18)` retrieves all students older than 18.

💡 **Interview Insight:** Expect questions like: *"How does Spring Data JPA create queries automatically?"* You should answer that it uses **method naming conventions** to translate method names into queries.

5. Performing CRUD Operations

Spring Data JPA makes CRUD operations straightforward.

Create

```
Student s = new Student();
s.setName("Alice");
s.setEmail("alice@gmail.com");
s.setAge(21);
studentRepository.save(s);
```

Read

```
List<Student> students = studentRepository.findAll();
```

Student student = studentRepository.findById(1L).orElse(null);

Update

Student student = studentRepository.findById(1L).get();
student.setName("Alice Updated");
studentRepository.save(student);

Delete

studentRepository.deleteById(1L);

💡 **Real-Time Example:** In an online course platform, CRUD operations allow admins to add new students, fetch registered students, update their information, or remove records.

6. Writing JPQL and Native Queries

Spring Data JPA supports **JPQL (Java Persistence Query Language)** and **native SQL queries**.

JPQL Example

@Query("SELECT s FROM Student s WHERE s.age > :age")
List<Student> getStudentsOlderThan(@Param("age") int age);

Here, the query works with entity objects, not database tables directly.

Native SQL Example

@Query(value = "SELECT * FROM students WHERE email = ?1", nativeQuery = true)
Student findByEmailNative(String email);

💡 **Real-Time Example:** Use JPQL for standard entity-based queries. Use native queries when working with vendor-specific functions or complex joins.

7. ORM Principles in Action

Object Relational Mapping (ORM) allows developers to think in terms of Java objects instead of tables and rows.

For example, when we execute:

```
Student student = new Student("Mike", "mike@gmail.com", 20);
studentRepository.save(student);
```

Spring Data JPA automatically generates SQL:

```
INSERT INTO students (name, email, age) VALUES ('Mike', 'mike@gmail.com', 20);
```

💡 **Industry Relevance:** ORM reduces the impedance mismatch between object-oriented Java code and relational databases. This is critical for building large-scale applications like e-commerce systems.

8. Layered Architecture for Data Access

Spring Boot applications typically follow a **layered architecture**:

- **Controller Layer** → Handles HTTP requests.
- **Service Layer** → Contains business logic.
- **Repository Layer** → Manages database operations.

Example

```
@RestController
@RequestMapping("/students")
public class StudentController {

    @Autowired
    private StudentService studentService;

    @PostMapping
    public Student saveStudent(@RequestBody Student student) {
        return studentService.saveStudent(student);
    }
}

@Service
public class StudentService {

    @Autowired
    private StudentRepository studentRepository;

    public Student saveStudent(Student student) {
        return studentRepository.save(student);
    }
}
```

This separation of concerns ensures that each layer has a clear responsibility, making applications **more maintainable and testable**.

Summary

- **Spring Data JPA** reduces boilerplate database code by using repositories.
- **Entities** map Java objects to database tables.
- **Repositories** provide CRUD and custom query methods.
- **JPQL and native SQL** allow flexible query definitions.
- **H2 vs MySQL** → H2 is ideal for testing, MySQL for production.
- **Layered architecture** ensures clean separation of concerns.

Important Questions

1. How does Spring Data JPA differ from plain Hibernate?
2. What are the advantages of using `JpaRepository`?

3. When would you prefer native queries over JPQL?
4. How does `@Entity` mapping work in ORM?
5. Explain the role of Service vs Repository layers.