LESSON 23
MySQL & JPA Basics

WEEK 05

What is JPA?

- **Definition**:
  - Java Persistence API (JPA) is a specification for object-relational mapping (ORM).
  - Maps Java objects to database tables.
- **Key Components**:
  - Entity, EntityManager, Persistence Unit.
- **Why Use JPA?**:
  - Simplifies database operations (no manual SQL for basic CRUD).
  - Supports multiple databases (e.g., MySQL, PostgreSQL).

Spring Data JPA Overview

- **What is Spring Data JPA?**:
  - Extension of Spring Data for JPA-based repositories.
  - Provides built-in methods for CRUD operations.
- **Key Features**:
  - Repository interfaces (CrudRepository, JpaRepository).
  - Query methods derived from method names.
  - Custom queries with @Query annotation.
- **Benefits**:
  - Reduces boilerplate code for database access.
  - Integrates seamlessly with Spring Boot.

Setting Up the Development Environment

- **Tools Required**:
  - JDK 17+, IntelliJ IDEA (or Eclipse or VS Code), MySQL, Maven/Gradle.
- **Steps**:
  - Install MySQL and create a database (e.g., school_db).
  - Configure IDE with Spring Boot plugin.
  - Add Spring Boot Starter dependencies.
- **Dependencies**:
  - spring-boot-starter-data-jpa
  - mysql-connector-java
- **Reference**: Spring Initializr

Creating a Spring Boot Project

- **Using Spring Initializr**:
  - Select Java, Gradle, Spring Boot 3.x.
  - Add dependencies: Spring Web, Spring Data JPA, MySQL Driver.
- **Project Structure**:
  - src/main/java: Application code.
  - src/main/resources: Configuration files (e.g., application.properties).
- **Example**:
  - Generate project at start.spring.io.
  - Import into IDE and run.

Configuring MySQL in Spring Boot

- **Configuration File**:
  - Edit application.properties to connect to MySQL.
- **Example Configuration**:

```
spring.datasource.url=jdbc:mysql://localhost:3306/school_db
spring.datasource.username=root
spring.datasource.password=your_password
spring.jpa.hibernate.ddl-auto=update
```

- **Explanation**:
  - ddl-auto=update: Automatically creates/updates database schema based on entities.
  - Ensure MySQL server is running.

Creating a JPA Entity

- **What is an Entity?**:
  - A Java class mapped to a database table.
- **Annotations**:
  - @Entity: Marks class as an entity.
  - @Id: Defines primary key.
  - @GeneratedValue: Auto-generates ID values.

Creating a JPA Repository

- **Repository Interface**:
  - Extends JpaRepository<EntityClass, IDType>.
  - Provides built-in CRUD methods.
- Example:

```
public interface StudentRepository extends JpaRepository<Student, Long> {
    // Custom query methods
}
```

- **Built-in Methods**:

  save(), findById(), findAll(), deleteById().

Implementing Create Operation

- **Purpose**:
  - Save a new entity to the database.
- **Explanation**:
  - save() persists the entity to the database.
  - Returns the saved entity with generated ID.

```
@Autowired
private StudentRepository repository;

public Student createStudent(Student student) {
    return repository.save(student);
}
```

Implementing Read Operation

- **Purpose**:
  - Retrieve entities from the database.
- **Explanation**:
  - findAll(): Retrieves all records.
  - findById(): Retrieves a single record by ID.

```
public List<Student> getAllStudents() {
    return repository.findAll();
}

public Optional<Student> getStudentById(Long id) {
    return repository.findById(id);
}
```

Implementing Update Operation

- **Purpose**:
  - Modify an existing entity in the database.
- **Explanation**:
  - Fetch entity, update fields, and save.

```
public Student updateStudent(Long id, Student updatedStudent) {
    Student student = repository.findById(id).orElseThrow();
    student.setName(updatedStudent.getName());
    student.setEmail(updatedStudent.getEmail());
    return repository.save(student);
}
```

Implementing Delete Operation

- **Purpose**:
  - Remove an entity from the database.
- **Explanation**:
  - deleteById() removes the entity with the specified ID.
  - Throws exception if ID does not exist.

```
public void deleteStudent(Long id) {
    repository.deleteById(id);
}
```

Creating a REST Controller

- **Purpose**:
  - Expose CRUD operations via RESTful APIs.

```java
@RestController
@RequestMapping("/api/students")
public class StudentController {
    @Autowired
    private StudentService service;

    @PostMapping
    public Student create(@RequestBody Student student) {
        return service.createStudent(student);
    }

    @GetMapping
    public List<Student> getAll() {
        return service.getAllStudents();
    }
}
```

Full CRUD Example

- **Scenario**:
  - Manage student records (create, read, update, delete).

```java
@RestController
@RequestMapping("/api/students")
public class StudentController {
    @Autowired
    private StudentService service;

    @PostMapping
    public Student create(@RequestBody Student student) {
        return service.createStudent(student);
    }

    @GetMapping("/{id}")
    public Student getById(@PathVariable Long id) {
        return service.getStudentById(id).orElseThrow();
    }

    @PutMapping("/{id}")
    public Student update(@PathVariable Long id, @RequestBody Student student) {
        return service.updateStudent(id, student);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        service.deleteStudent(id);
    }
}
```

Testing APIs with Postman

- **Steps**:
  - Start Spring Boot application.
  - Use Postman to send HTTP requests (POST, GET, PUT, DELETE).
- **Example**:
  - POST: http://localhost:8080/api/students with JSON body:

```
{"name": "John Doe", "email": "john@example.com"}
```

Validation with JPA

- **Purpose**:
  - Ensure valid data before saving to database.
- **Explanation**:
  - Use annotations like @NotNull, @Email from javax.validation.

```
public class Student {
    @NotNull
    private String name;

    @Email
    private String email;
}
```

Introduction to JPA Relationship Annotations

- **Purpose**:
  - Define relationships between entities (e.g., Student, Department) in JPA.
  - Annotations: @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.
- **Key Concepts**:
  - Owning side: Defines the relationship (owns the foreign key).
  - Inverse side: References the owning side (uses mappedBy).
  - Cascade and fetch strategies control behavior and performance.
- **Why Important?**:
  - Enables modeling of complex data relationships in the database.
  - Simplifies querying and data management.

Configuring @OneToOne Relationship

- **Definition**:
  - One entity instance is related to exactly one instance of another entity.
  - Example: A Student has one Address.
- **Explanation**:
  - @JoinColumn: Specifies the foreign key column (address_id) in the Student table.
  - cascade: Propagates operations (e.g., save, delete) to the related entity.

```java
@Entity
public class Student {
    @Id
    @GeneratedValue
    private Long id;
    private String name;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id")
    private Address address;
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;
    private String street;
}
```

Configuring @OneToMany and @ManyToOne

- **Definition**:
  - @OneToMany: One entity relates to multiple instances of another (e.g., one Department has many Students).
  - @ManyToOne: Many instances relate to one instance (e.g., many Students belong to one Department).
- **Explanation**:
  - @ManyToOne (owning side): Defines the foreign key (department_id) in the Student table.
  - @OneToMany (inverse side): Uses mappedBy to reference the owning side.

```java
@Entity
public class Department {
    @Id
    @GeneratedValue
    private Long id;
    private String name;

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
    private List<Student> students = new ArrayList<>();
}

@Entity
public class Student {
    @Id
    @GeneratedValue
    private Long id;
    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
}
```

Configuring @ManyToMany Relationship

- **Definition**:
  - Multiple instances of one entity relate to multiple instances of another.
  - Example: Students enroll in multiple Courses, and Courses have multiple Students.
- **Explanation**:
  - @JoinTable: Defines the join table (student_course) with foreign keys.
  - mappedBy: Specifies the owning side (Student) to avoid duplicate mappings.

```java
@Entity
public class Student {
    @Id
    @GeneratedValue
    private Long id;
    private String name;

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses = new ArrayList<>();
}

@Entity
public class Course {
    @Id
    @GeneratedValue
    private Long id;
    private String title;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students = new ArrayList<>();
}
```

Cascade and Fetch Strategies in Relationships

- **Cascade**:
  - Controls propagation of operations (e.g., save, delete) to related entities.
  - Options: CascadeType.ALL, PERSIST, MERGE, REMOVE, etc.
  - Example: cascade = CascadeType.ALL saves related entities automatically.
- **Fetch Strategies**:
  - FetchType.LAZY: Loads related data only when accessed (default for @OneToMany, @ManyToMany).
  - FetchType.EAGER: Loads related data immediately (default for @ManyToOne, @OneToOne).

```
@OneToMany(mappedBy = "department", cascade = CascadeType.PERSIST, fetch = FetchType.LAZY)
private List<Student> students;
```

Introduction to Joins in JPA

- **What are Joins in JPA?**:
  - Joins in JPA are used to combine data from multiple entities based on relationships.
  - Defined in JPQL (Java Persistence Query Language) or Criteria API.
  - Support for INNER JOIN, LEFT JOIN, RIGHT JOIN, and implicit joins.
- **Why Use Joins?**:
  - Retrieve related data in a single query, avoiding multiple database calls.
  - Essential for querying associations like @OneToMany or @ManyToMany.

Inner Join in JPA

- **Definition**:
  - Returns records that have matching values in both entities.
  - Equivalent to SQL INNER JOIN.
- **Explanation**:
  - Joins Student and Department entities on their relationship.
  - Only includes students with a matching department.

```
@Query("SELECT s FROM Student s JOIN s.department d WHERE d.name = :deptName")
List<Student> findStudentsByDepartment(@Param("deptName") String deptName);
```

Left Outer Join in JPA

- **Definition**:
  - Returns all records from the left entity and matching records from the right.
  - Non-matching right records are null.
- **Explanation**:
  - Includes all students, even those without a department.
  - Useful for optional relationships.

```
@Query("SELECT s FROM Student s LEFT JOIN s.department d WHERE d.name = :deptName OR d IS NULL")
List<Student> findStudentsWithOptionalDepartment(@Param("deptName") String deptName);
```

Right Outer Join in JPA

- **Definition**:
  - Returns all records from the right entity and matching records from the left.
  - Non-matching left records are null.
- **Explanation**:
  - Includes all departments, even those without students.
  - Less common than LEFT JOIN but symmetric.

```
@Query("SELECT d FROM Department d RIGHT JOIN d.students s WHERE s.name = :studentName")
List<Department> findDepartmentsByStudent(@Param("studentName") String studentName);
```

Right Outer Join in JPA

- **Definition**:
  - Returns all records from the right entity and matching records from the left.
  - Non-matching left records are null.

```
@Query("SELECT d FROM Department d RIGHT JOIN d.students s WHERE s.name = :studentName")
List<Department> findDepartmentsByStudent(@Param("studentName") String studentName);
```

Fetch Joins in JPA

- **Definition**:
  - Eagerly fetches related entities in a single query to avoid N+1 problem.
  - Uses FETCH keyword in JPQL.
- **Explanation**:
  - Loads departments immediately, preventing lazy loading exceptions.
  - Improves performance for read operations.

```
@Query("SELECT s FROM Student s JOIN FETCH s.department d")
List<Student> findAllStudentsWithDepartments();
```

Introduction to Paging in JPA

- **Introduction to Paging in JPA**
- **What is Paging?**:
  - Paging allows retrieving large datasets in smaller chunks (pages) to improve performance and usability.
  - Essential for applications with large databases to avoid loading all data at once.
- **Key Components in Spring Data JPA**:
  - Pageable: Interface for pagination and sorting information.
  - Page: Represents a page of data with metadata (total pages, total elements).
  - Slice: Similar to Page but without total count (faster for large datasets).
- **Why Use Paging?**:
  - Reduces memory usage, improves response times, and enables features like infinite scrolling.

Using Pageable in JpaRepository

- **Pageable Interface**:
  - Created using PageRequest.of(pageNumber, pageSize, sort) to specify page index, size, and sorting.
- **Repository Methods**:
  - Extend JpaRepository and add methods returning Page<T> or Slice<T>.

```
public interface StudentRepository extends JpaRepository<Student, Long> {
    Page<Student> findAll(Pageable pageable);
}
```

```
Pageable pageable = PageRequest.of(0, 10, Sort.by("name").ascending());
Page<Student> studentsPage = repository.findAll(pageable);
```

Working with Page and Slice

- **Page**:
  - Provides full pagination info: content, total pages, total elements.
  - Example: studentsPage.getTotalElements(), studentsPage.getTotalPages().
- **Slice**:
  - Lighter than Page; no total count (avoids expensive COUNT queries).
  - Use for "load more" features where total is not needed.

```
Slice<Student> studentsSlice = repository.findAll(Pageable pageable);
List<Student> content = studentsSlice.getContent();
boolean hasNext = studentsSlice.hasNext();
```

Custom Queries with Paging

- **Custom JPQL Queries**:
  - Use @Query with Pageable for custom pagination.
- **Explanation**:
  - Pageable is appended as the last parameter in custom queries.
  - Supports sorting and pagination on derived or custom queries.

```
public interface StudentRepository extends JpaRepository<Student, Long> {
    @Query("SELECT s FROM Student s WHERE s.name LIKE %:name%")
    Page<Student> findByNameContaining(@Param("name") String name, Pageable pageable);
}
```

```
Page<Student> results = repository.findByNameContaining("John", pageable);
```

Best Practices for Paging in JPA

- **Key Practices**:
  - Use Slice for large datasets to avoid slow COUNT queries.
  - Combine with sorting: Sort.by("field").ascending() for user-friendly results.
  - Handle edge cases: Empty pages, invalid page numbers.
  - Use in REST APIs: Return Page metadata in responses for client-side pagination.
- **Common Pitfalls**:
  - N+1 queries: Use Fetch Joins with paging for relationships.
  - Performance: Index columns used in sorting/filters.

Introduction to Inheritance in JPA Entities

- **Introduction to Inheritance in JPA Entities**
- **What is Inheritance in JPA?**:
  - JPA supports inheritance to model hierarchical entity classes, mapping Java OO inheritance to relational databases.
  - Allows subclasses to inherit fields and relationships from a superclass.
- **Inheritance Strategies**:
  - @Inheritance(strategy = InheritanceType.SINGLE_TABLE): All classes in one table with discriminator.
  - @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS): Separate table per concrete class.
  - @Inheritance(strategy = InheritanceType.JOINED): Separate table for superclass and each subclass (1:1 relationship via shared primary key).
- **Focus on 1:1 Inheritance**:
  - Refers to JOINED strategy, where subclass tables link 1:1 to superclass table using shared PK.
- **Reference**: JPA Inheritance Specification

JOINED Inheritance Strategy (1:1 Mapping)

- **Definition**:
  - Superclass has its own table; each subclass has a separate table with only subclass-specific fields.
  - Subclass tables reference superclass table via shared primary key (1:1 relationship).
- **Annotations**:
  - @Inheritance(strategy = InheritanceType.JOINED) on superclass.
  - @PrimaryKeyJoinColumn optional for customizing join column.
- **Database Structure**:
  - Superclass table: Common fields + PK.
  - Subclass table: Subclass fields + PK (foreign key to superclass PK).
- **Explanation**:
  - Queries join tables as needed; supports polymorphism (e.g., querying superclass returns mixed subclass instances).

Example of 1:1 Inheritance in JPA

- **Generated Tables**:
  - person: id (PK), name.
  - student: id (PK/FK to person.id), major.
  - teacher: id (PK/FK to person.id), subject.
- **Explanation**:
  - Inserting a Student creates rows in both person and student tables with same id.
  - Query: SELECT p FROM Person p joins tables to fetch mixed Student/Teacher instances.
- **Reference**: Thorben Janssen: JPA Joined Strategy

```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    // Getters and setters
}

@Entity
public class Student extends Person {
    private String major;
    // Getters and setters
}

@Entity
public class Teacher extends Person {
    private String subject;
    // Getters and setters
}
```

Advantages and Disadvantages of 1:1 Inheritance

- **Advantages**:
  - Normalized database: No redundant fields; easy to add new subclasses.
  - Supports polymorphism: Queries on superclass return subclass instances.
  - Efficient for reads on specific subclasses (no unnecessary joins).
- **Disadvantages**:
  - Performance overhead: Joins required for superclass queries.
  - Complex inserts/updates: Multiple tables involved.
  - Not suitable for deep hierarchies due to join complexity.
- **When to Use**:
  - When normalization is important and hierarchies are not too deep.

Best Practices for 1:1 Inheritance in JPA

- **Key Practices**:
  - Use @DiscriminatorColumn if needed for explicit type discrimination (though optional in JOINED).
  - Optimize queries with Fetch Joins to avoid N+1 issues.
  - Index join columns for better performance.
  - Test polymorphism: Ensure repositories handle superclass queries correctly.
- **Common Pitfalls**:
  - Overusing joins in deep hierarchies leading to slow queries.
  - Forgetting to generate IDs in superclass.

```
@Query("SELECT p FROM Person p JOIN FETCH p WHERE p.id = :id")
Person findByIdWithFetch(@Param("id") Long id);
```

Conclusion and Next Steps

- **Summary**:
  - Learned to build a CRUD application with Spring Boot, JPA, and MySQL.
  - Covered entities, repositories, REST APIs, and basics JPA features.
- **Next Steps**:
  - Build a full-stack application with a front-end (e.g., React).
- **References**:
  - Spring Boot
  - Spring Data JPA
  - JPA Specification