### **Exercise 1: Employee Management System - Overview and Setup**

### **Step 1: Create the Employee Entity**

1. **Create a new Java class named Employee** in the appropriate package (e.g., com.example.employeemanagementsystem.entity).
2. **Define the fields:**
   * id: Long (generated value)
   * firstName: String
   * lastName: String
   * email: String
   * department: Department (relationship to Department entity)
3. **Annotate the class and fields:**
   * Use @Entity to mark the class as a JPA entity.
   * Use @Id to mark the id field as the primary key.
   * Use @GeneratedValue to specify the ID generation strategy (e.g., GenerationType.IDENTITY).
   * Use @Column for optional column-specific mappings.
   * Use @ManyToOne for the relationship with the Department (many employees belong to one department).

### **Step 2: Create the Department Entity**

1. **Create a new Java class named Department** in the same package as the Employee entity.
2. **Define the fields:**
   * id: Long (generated value)
   * departmentName: String
   * location: String
3. **Annotate the class and fields:**
   * Use @Entity to mark the class as a JPA entity.
   * Use @Id to mark the id field as the primary key.
   * Use @GeneratedValue to specify the ID generation strategy.
   * Use @Column for optional column-specific mappings.
   * Use @OneToMany for the relationship with an Employee (one department has many employees).

### **Code Example:**

Java

import jakarta.persistence.\*;

import lombok.Data;

@Entity

@Data

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String firstName;

private String lastName;

private String email;

@ManyToOne

private Department department;

}

import jakarta.persistence.\*;

import lombok.Data;

@Entity

@Data

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String

departmentName;

private String location;

@OneToMany(mappedBy = "department")

private List<Employee> employees = new ArrayList<>();

}

**2. Employee Management System - Creating Entities**

### **Creating the Entities**

Java

import jakarta.persistence.\*;

import lombok.Data;

@Entity

@Data

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String

email;

@ManyToOne

private Department department;

}

Use code with caution.

Java

import jakarta.persistence.\*;

import lombok.Data;

@Entity

@Data

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "department")

private List<Employee> employees = new ArrayList<>();

}

### **Explanation of the Code**

* **Annotations:**
  + @Entity: Indicates that a class is a JPA entity, mapped to a database table.
  + @Id: Marks a field as the primary key.
  + @GeneratedValue: Specifies how the primary key is generated (in this case, auto-increment).
  + @ManyToOne: Defines a many-to-one relationship, indicating that an Employee belongs to one Department.
  + @OneToMany: Defines a one-to-many relationship, indicating that a Department can have many Employees.
  + mappedBy: Specifies the owning side of the relationship, in this case, the Employee entity owns the relationship.
* **Fields:**
  + Employee: id, name, email, and a department reference.
  + Department: id, name, and a List of Employee objects.

### **Key Points**

* The Employee entity has a department field of type Department, indicating that each employee belongs to one department.
* The Department entity has a List of employees, indicating that a department can have multiple employees.
* The mappedBy attribute in @OneToMany specifies the owning side of the relationship, which is important for JPA to manage the relationship correctly.

With these entities, we've defined the basic structure of our Employee Management System. In the next steps, we'll create repositories to interact with these entities.

**3. Employee Management System - Creating Repositories:**

### **Understanding Spring Data Repositories**

Spring Data JPA provides a significant abstraction over JPA repositories. By extending Spring Data's repository interfaces, you can perform CRUD operations and create custom query methods with minimal effort.

**Benefits:**

* **Reduced boilerplate code:** No need to write repetitive data access logic.
* **Simplified query creation:** Leverage method naming conventions to create dynamic queries.
* **Integration with Spring ecosystem:** Seamlessly works with other Spring components.

### **Creating Repositories**

Let's create repository interfaces for Employee and Department entities.

Java

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

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Custom query methods can be defined here

}

public interface DepartmentRepository extends JpaRepository<Department, Long> {

// Custom query methods can be defined here

}

Use code with caution.

* **Explanation:**
  + JpaRepository provides basic CRUD operations for entities.
  + The first type parameter is the entity class, and the second is the ID type.
  + We'll define custom query methods as needed within these interfaces.

### **Derived Query Methods**

Spring Data JPA supports creating query methods based on method names.

**Example:**

Java

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByDepartmentName(String departmentName);

List<Employee> findByFirstNameIgnoreCase(String firstName);

}

Use code with caution.

* **Explanation:**
  + findByDepartmentName: Finds employees by department name.
  + findByFirstNameIgnoreCase: Finds employees by first name, ignoring case.

Spring Data will automatically create the corresponding SQL query based on the method name.

**4. Employee Management System - Implementing CRUD Operations**

### **Creating Services**

Before implementing controllers, let's create service layers to encapsulate business logic:

Employee.java

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

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public List<Employee> getAllEmployees() {

return employeeRepository.findAll();

}

public Employee getEmployeeById(Long id) {

return employeeRepository.findById(id).orElse(null);

}

public Employee createEmployee(Employee employee) {

return employeeRepository.save(employee);

}

public Employee updateEmployee(Employee employee) {

return employeeRepository.save(employee);

}

public void deleteEmployee(Long id) {

employeeRepository.deleteById(id);

}

}

Similarly, create a DepartmentService with corresponding methods.

### **Creating Controllers**

Now, let's create REST controllers to expose CRUD operations:

Java

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

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

import java.util.List;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@GetMapping

public ResponseEntity<List<Employee>> getAllEmployees() {

List<Employee> employees = employeeService.getAllEmployees();

return ResponseEntity.ok(employees);

}

@GetMapping("/{id}")

public ResponseEntity<Employee>

getEmployeeById(@PathVariable Long id) {

Employee employee = employeeService.getEmployeeById(id);

if (employee != null) {

return ResponseEntity.ok(employee);

} else {

return ResponseEntity.notFound().build();

}

}

@PostMapping

public ResponseEntity<Employee> createEmployee(@RequestBody Employee employee) {

Employee createdEmployee = employeeService.createEmployee(employee);

return

ResponseEntity.status(HttpStatus.CREATED).body(createdEmployee);

}

@PutMapping("/{id}")

public ResponseEntity<Employee> updateEmployee(@PathVariable Long id, @RequestBody Employee employee)

{

employee.setId(id);

Employee updatedEmployee = employeeService.updateEmployee(employee);

return ResponseEntity.ok(updatedEmployee);

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteEmployee(@PathVariable Long id) {

employeeService.deleteEmployee(id);

return ResponseEntity.noContent().build();

}

}

**5. Employee Management System - Defining Query Methods:**

### **Understanding the Requirements**

We'll enhance our repositories to support custom queries using both derived query methods and @Query annotations. We'll also explore named queries.

### **Defining Query Methods**

We can leverage Spring Data JPA's ability to create queries based on method names:

Java

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

public interface EmployeeRepository extends JpaRepository<Employee,

Long> {

List<Employee> findByDepartmentName(String

departmentName);

List<Employee> findByFirstNameIgnoreCase(String firstName);

List<Employee> findBySalaryGreaterThan(Double salary);

}

For more complex queries, we can use the @Query annotation:

Java

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

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

public interface

EmployeeRepository extends JpaRepository<Employee, Long> {

@Query("SELECT e FROM Employee e WHERE e.department.name = :departmentName")

List<Employee> getEmployeesByDepartmentName(@Param("departmentName") String departmentName);

}

### **Named Queries**

Named queries are pre-defined queries that can be reused:

Java

import jakarta.persistence.NamedQuery;

@Entity

@NamedQuery(name = "Employee.findByDepartment", query = "SELECT e FROM Employee e WHERE e.department.name = :departmentName")

public class Employee {

// ...

}

To use a named query in the repository:

Java

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

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

import org.springframework.data.repository.query.Param;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

@Query(name = "Employee.findByDepartment")

List<Employee> getEmployeesByDepartmentName(@Param("departmentName") String departmentName);

}

### **Additional Considerations**

* **Native Queries:** For performance or database-specific features, use @NativeQuery with nativeQuery = true.
* **Paging and Sorting:** Use Pageable and Sort interfaces for pagination and sorting.
* **Specifications:** For dynamic query building, consider using Spring Data JPA Specifications.

### **Example**

Java

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.jpa.domain.Specification;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

Page<Employee> findByDepartmentName(String departmentName, Pageable pageable);

List<Employee> findAll(Specification<Employee> spec);

}

By combining these techniques, you can create versatile and efficient query methods for your Employee Management System.

**6. Employee Management System - Implementing Pagination and Sorting:**

### **Pagination**

Spring Data JPA provides built-in support for pagination using Page and Pageable interfaces.

Java

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

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

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

Page<Employee> findAll(Pageable pageable);

}

In the controller:

Java

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@GetMapping

public Page<Employee> getEmployees(Pageable pageable) {

return employeeService.getEmployees(pageable);

}

}

### **Sorting**

Spring Data JPA also supports sorting using the Sort interface.

Java

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.domain.Sort;

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

public interface EmployeeRepository extends JpaRepository<Employee,

Long> {

Page<Employee> findAll(Pageable pageable);

List<Employee> findAll(Sort sort);

}

In the controller:

Java

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.domain.Sort;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@GetMapping

public Page<Employee> getEmployees(Pageable pageable, @RequestParam(required = false) String sort) {

if (sort != null) {

pageable = Pageable.of(pageable.getPageNumber(), pageable.getPageSize(), Sort.by(sort));

}

return employeeService.getEmployees(pageable);

}

}

### **Combining Pagination and Sorting**

We've combined pagination and sorting in the controller. The Pageable interface automatically handles page number, page size, and sort information.

**7. Employee Management System - Enabling Entity Auditing:**

#### **1. Configuration**

Spring Data JPA provides built-in support for auditing. We need to configure it in our application.properties:

Properties:

spring.jpa.auditing.enabled=true

spring.jpa.auditing.is-always-default-auditor=true

* spring.jpa.auditing.enabled: Enables auditing.
* spring.jpa.auditing.is-always-default-auditor: Sets a default auditor.

#### **2. Entity Annotations**

We'll add the following annotations to our entities:

Java

import jakarta.persistence.\*;

import org.springframework.data.annotation.CreatedBy;

import org.springframework.data.annotation.CreatedDate;

import org.springframework.data.annotation.LastModifiedBy;

import org.springframework.data.annotation.LastModifiedDate;

import org.springframework.data.jpa.domain.support.Auditable;

import java.time.Instant;

@Entity

@Data

public class Employee implements Auditable<String> {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

@ManyToOne

private Department department;

@CreatedBy

private String createdBy;

@CreatedDate

private Instant createdDate;

@LastModifiedBy

private String lastModifiedBy;

@LastModifiedDate

private Instant lastModifiedDate;

}

Similarly, add the same annotations to the Department entity.

### **How it Works**

* @CreatedBy and @LastModifiedBy: These annotations store the username of the user who created or modified the entity.
* @CreatedDate and @LastModifiedDate: These annotations store the timestamp of creation or modification.
* Auditable<String>: This interface indicates that the entity supports auditing. The generic type String specifies the type of the user's identifier.

**8. Employee Management System - Creating Projections:**

### **Interface-Based Projections**

Java

public interface EmployeeProjection {

Long getId();

String getName();

String getEmail();

}

In the repository:

Java

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<EmployeeProjection> findAll();

}

### **Class-Based Projections**

Java

public class EmployeeProjection {

private Long id;

private String name;

private String email;

// Constructors, getters, and setters

}

In the repository:

Java

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<EmployeeProjection> findAll();

}

### **Using @Value and Constructor Expressions**

For more complex projections, we can use @Value and constructor expressions:

Java

public interface EmployeeProjection {

@Value("#{target.id + '-' + target.name}")

String getFormattedName();

}

### **Additional Considerations**

* **Dynamic Projections:** Spring Data JPA also supports dynamic projections for runtime-defined projections.
* **Performance:** Consider performance implications when using projections.
* **N+1 Problem:** Be aware of potential N+1 issues and use appropriate fetching strategies.

### **Example**

Java

public interface EmployeeSummaryProjection {

Long getId();

String getName();

String getDepartmentName(); // Assuming Department has a name property

}

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<EmployeeSummaryProjection> findByDepartmentName(String departmentName);

}

By using projections, we can optimize data retrieval and tailor responses to specific requirements.

**9. Employee Management System - Customizing Data Source Configuration:**

### **Spring Boot Auto-Configuration for Data Sources**

Spring Boot auto-configuration simplifies data source setup. By including the appropriate JDBC driver library on your classpath, Spring Boot automatically configures a basic data source based on properties you provide.

### **Externalizing Configuration with application.properties**

Properties

# Database configuration

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.username=sa

spring.datasource.password= This configuration defines a connection to an in-memory H2 database. You can modify these properties to connect to different databases like MySQL, PostgreSQL, etc.

### **Managing Multiple Data Sources**

Spring Boot allows you to configure multiple data sources for different purposes. Here's an example:

Properties

# Primary data source for employees

spring.datasource.primary.url=jdbc:h2:mem:testdb

spring.datasource.primary.username=sa

spring.datasource.primary.password=

# Secondary data source for reports

spring.datasource.secondary.url=jdbc:mysql://localhost:3306/reportsdb

spring.datasource.secondary.username=reportuser

spring.datasource.secondary.password=reportpassword

Spring Boot automatically creates a DataSource bean for each named configuration (primaryDataSource and secondaryDataSource in this case).

**Specifying the Data Source:**

* Use @Transactional annotation and specify the data source qualifier:

Java

@Transactional(dataSource = "primaryDataSource")

public void saveEmployee(Employee employee) {

// ...

}

* Use the JdbcTemplate class with the desired data source bean:

Java

JdbcTemplate jdbcTemplate = new JdbcTemplate(secondaryDataSource);

jdbcTemplate.execute("...");

**10. Employee Management System - Hibernate-Specific Features:**

### **1. Hibernate-Specific Annotations**

Hibernate offers additional annotations beyond JPA for fine-grained control over entity mappings:

* **@GenericGenerator:** For custom ID generation strategies.
* **@Where:** To apply filters to queries.
* **@Formula:** To map database expressions to entity properties.
* **@Fetch:** To control entity fetching strategies (eager, lazy, join).
* **@BatchSize:** To optimize batch fetching.
* **@OptimisticLocking:** To implement optimistic locking.

**Example:**

Java

import org.hibernate.annotations.GenericGenerator;

@Entity

public class Employee {

@Id

@GeneratedValue(generator = "uuid-generator")

@GenericGenerator(name = "uuid-generator", strategy = "uuid2")

private String id;

// ... other fields

}

### **2. Configuring Hibernate Dialect and Properties**

The Hibernate dialect defines database-specific SQL syntax and features. Proper configuration can significantly improve performance:

Properties

spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect

spring.jpa.hibernate.ddl-auto=update

spring.jpa.hibernate.jdbc.batch\_size=20

spring.jpa.hibernate.order\_inserts=true

spring.jpa.hibernate.order\_updates=true

* **Dialect:** Specifies the database type (replace with your dialect).
* **DDL Auto:** Defines how schema is handled (update, create, validate, none).
* **Batch size:** Sets the number of SQL statements to batch together.
* **Order inserts and updates:** Improves performance by ordering SQL statements.

### **3. Batch Processing**

Hibernate supports batching for improved performance when inserting or updating multiple entities:

Java

@Transactional

public void saveEmployees(List<Employee> employees) {

for (Employee employee : employees) {

employeeRepository.save(employee);

}

}

With batching enabled, Hibernate will execute multiple insert statements in a single batch, reducing database round trips.

**Additional Considerations:**

* **Second-level cache:** Consider using Hibernate's second-level cache to improve read performance.
* **Performance tuning:** Analyze query performance and optimize accordingly.
* **Testing:** Thoroughly test changes to avoid unexpected behavior.

By effectively using Hibernate-specific features, you can optimize your application's performance and data management capabilities.