Name: Kruthi Venkatesh

Email: 125156165@sastra.ac.in

# WEEK 4 EXERCISES

Exercise 2: Employee Management System - Creating Entities

In this exercise, you'll define JPA entities for Employee and Department with the necessary fields and relationships.

### 1. **Creating JPA Entities**

#### 1.1 **Define the** Employee **Entity**

The Employee entity will represent the employee data with fields like id, name, email, and department.

1. Create a class named Employee in the model package.

java

Copy code

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

import lombok.Data;

import lombok.NoArgsConstructor;

import lombok.AllArgsConstructor;

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(nullable = false)

private String name;

@Column(nullable = false, unique = true)

private String email;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "department\_id")

private Department department;

}

* **Explanation**:
  + @Entity: Marks the class as a JPA entity.
  + @Table(name = "employees"): Specifies the table name.
  + @Id and @GeneratedValue(strategy = GenerationType.IDENTITY): Marks the primary key and specifies that the ID should be generated automatically.
  + @Column: Specifies column properties.
  + @ManyToOne: Establishes a many-to-one relationship with the Department entity.
  + @JoinColumn(name = "department\_id"): Specifies the foreign key column in the Employee table.

#### 1.2 **Define the** Department **Entity**

The Department entity will represent department data with fields like id and name.

1. Create a class named Department in the model package.

java

Copy code

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

import lombok.Data;

import lombok.NoArgsConstructor;

import lombok.AllArgsConstructor;

import java.util.List;

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name = "departments")

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(nullable = false, unique = true)

private String name;

@OneToMany(mappedBy = "department", cascade = CascadeType.ALL, fetch = FetchType.LAZY)

private List<Employee> employees;

}

* **Explanation**:
  + @Entity: Marks the class as a JPA entity.
  + @Table(name = "departments"): Specifies the table name.
  + @Id and @GeneratedValue(strategy = GenerationType.IDENTITY): Marks the primary key and specifies that the ID should be generated automatically.
  + @Column: Specifies column properties.
  + @OneToMany(mappedBy = "department", cascade = CascadeType.ALL, fetch = FetchType.LAZY): Establishes a one-to-many relationship with the Employee entity. The mappedBy attribute indicates that the department field in the Employee entity owns the relationship. CascadeType.ALL ensures that any operation (like save or delete) performed on the Department entity will cascade to its related Employee entities.

### 2. **Mapping Entities to Database Tables**

By using the JPA annotations in the Employee and Department entities, you've successfully mapped these entities to the corresponding database tables. The relationships between the entities are also established:

* **Department to Employee**: One-to-Many
* **Employee to Department**: Many-to-One

### Final Code Structure

Your project should now have the following structure:

css

Copy code

src/main/java/com/example/employeemanagementsystem/

└── model/

├── Employee.java

└── Department.java

Exercise 3: Employee Management System - Creating Repositories

In this exercise, you'll create repositories for the Employee and Department entities, which will allow you to perform CRUD operations on the data.

**1. Overview of Spring Data Repositories**

**Spring Data Repositories** provide a convenient way to manage data access layers. Some benefits include:

* **Boilerplate Reduction**: Spring Data JPA automatically generates implementations for common CRUD operations, eliminating the need to write boilerplate code.
* **Derived Query Methods**: You can define query methods by simply declaring them in the repository interface. Spring Data JPA will derive the implementation based on the method name.
* **Custom Queries**: If needed, you can define custom queries using JPQL (Java Persistence Query Language) or native SQL.

**2. Creating Repositories**

You will now create the EmployeeRepository and DepartmentRepository interfaces.

**2.1 Create the EmployeeRepository**

1. Create a new package named repository under src/main/java/com/example/employeemanagementsystem.
2. Create an interface named EmployeeRepository in this package.

java

Copy code

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

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

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Derived query method to find employees by department

List<Employee> findByDepartmentId(Long departmentId);

// Derived query method to find an employee by email

Employee findByEmail(String email);

}

* **Explanation**:
  + extends JpaRepository<Employee, Long>: Inherits basic CRUD operations (like save, findById, findAll, delete) for the Employee entity.
  + findByDepartmentId(Long departmentId): A derived query method to find all employees belonging to a specific department.
  + findByEmail(String email): A derived query method to find an employee by their email address.

**2.2 Create the DepartmentRepository**

1. Create an interface named DepartmentRepository in the same repository package.

java

Copy code

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Department;

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

import org.springframework.stereotype.Repository;

@Repository

public interface DepartmentRepository extends JpaRepository<Department, Long> {

// Derived query method to find a department by name

Department findByName(String name);

}

* **Explanation**:
  + extends JpaRepository<Department, Long>: Inherits basic CRUD operations (like save, findById, findAll, delete) for the Department entity.
  + findByName(String name): A derived query method to find a department by its name.

**Final Code Structure**

Your project should now have the following structure:

css

Copy code

src/main/java/com/example/employeemanagementsystem/

├── model/

│ ├── Employee.java

│ └── Department.java

└── repository/

├── EmployeeRepository.java

└── DepartmentRepository.java

**Summary**

* **Spring Data Repositories** provide an easy way to implement the data access layer with minimal code.
* You created EmployeeRepository and DepartmentRepository interfaces that extend JpaRepository to perform CRUD operations.
* You defined derived query methods to fetch employees by department and to find departments by name.

Exercise 4: Employee Management System - Implementing CRUD Operations

In this exercise, you will implement CRUD operations for managing employees and departments using the JpaRepository methods and expose these operations via RESTful endpoints using EmployeeController and DepartmentController.

**1. Basic CRUD Operations**

**1.1 Create the Service Layer**

The service layer will handle the business logic for CRUD operations. You’ll create services for Employee and Department.

1. **Create a package named service** under src/main/java/com/example/employeemanagementsystem.
2. **Create the EmployeeService class** in this package:

java

Copy code

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

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;

// Create or update an employee

public Employee saveEmployee(Employee employee) {

return employeeRepository.save(employee);

}

// Get all employees

public List<Employee> getAllEmployees() {

return employeeRepository.findAll();

}

// Get employee by ID

public Optional<Employee> getEmployeeById(Long id) {

return employeeRepository.findById(id);

}

// Delete an employee by ID

public void deleteEmployee(Long id) {

employeeRepository.deleteById(id);

}

// Get employees by department ID

public List<Employee> getEmployeesByDepartmentId(Long departmentId) {

return employeeRepository.findByDepartmentId(departmentId);

}

}

1. **Create the DepartmentService class**:

java

Copy code

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Department;

import com.example.employeemanagementsystem.repository.DepartmentRepository;

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

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class DepartmentService {

@Autowired

private DepartmentRepository departmentRepository;

// Create or update a department

public Department saveDepartment(Department department) {

return departmentRepository.save(department);

}

// Get all departments

public List<Department> getAllDepartments() {

return departmentRepository.findAll();

}

// Get department by ID

public Optional<Department> getDepartmentById(Long id) {

return departmentRepository.findById(id);

}

// Delete a department by ID

public void deleteDepartment(Long id) {

departmentRepository.deleteById(id);

}

}

**2. Implement RESTful Endpoints**

Next, you'll create controllers to expose these CRUD operations via RESTful endpoints.

1. **Create a package named controller** under src/main/java/com/example/employeemanagementsystem.
2. **Create the EmployeeController class**:

java

Copy code

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.service.EmployeeService;

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

import org.springframework.http.ResponseEntity;

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

import java.util.List;

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

// Create or update an employee

@PostMapping

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

Employee savedEmployee = employeeService.saveEmployee(employee);

return ResponseEntity.ok(savedEmployee);

}

// Get all employees

@GetMapping

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

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

return ResponseEntity.ok(employees);

}

// Get employee by ID

@GetMapping("/{id}")

public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id) {

return employeeService.getEmployeeById(id)

.map(ResponseEntity::ok)

.orElse(ResponseEntity.notFound().build());

}

// Delete an employee by ID

@DeleteMapping("/{id}")

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

employeeService.deleteEmployee(id);

return ResponseEntity.noContent().build();

}

// Get employees by department ID

@GetMapping("/department/{departmentId}")

public ResponseEntity<List<Employee>> getEmployeesByDepartmentId(@PathVariable Long departmentId) {

List<Employee> employees = employeeService.getEmployeesByDepartmentId(departmentId);

return ResponseEntity.ok(employees);

}

}

1. **Create the DepartmentController class**:

java

Copy code

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.model.Department;

import com.example.employeemanagementsystem.service.DepartmentService;

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

import org.springframework.http.ResponseEntity;

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

import java.util.List;

@RestController

@RequestMapping("/api/departments")

public class DepartmentController {

@Autowired

private DepartmentService departmentService;

// Create or update a department

@PostMapping

public ResponseEntity<Department> createOrUpdateDepartment(@RequestBody Department department) {

Department savedDepartment = departmentService.saveDepartment(department);

return ResponseEntity.ok(savedDepartment);

}

// Get all departments

@GetMapping

public ResponseEntity<List<Department>> getAllDepartments() {

List<Department> departments = departmentService.getAllDepartments();

return ResponseEntity.ok(departments);

}

// Get department by ID

@GetMapping("/{id}")

public ResponseEntity<Department> getDepartmentById(@PathVariable Long id) {

return departmentService.getDepartmentById(id)

.map(ResponseEntity::ok)

.orElse(ResponseEntity.notFound().build());

}

// Delete a department by ID

@DeleteMapping("/{id}")

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

departmentService.deleteDepartment(id);

return ResponseEntity.noContent().build();

}

}

**3. Summary**

* **Service Layer**: The service layer handles the business logic and interacts with the repositories to perform CRUD operations.
* **Controllers**: The EmployeeController and DepartmentController expose RESTful endpoints to manage employees and departments.
* **Endpoints Overview**:
  + **EmployeeController**:
    - POST /api/employees: Create or update an employee.
    - GET /api/employees: Get all employees.
    - GET /api/employees/{id}: Get an employee by ID.
    - DELETE /api/employees/{id}: Delete an employee by ID.
    - GET /api/employees/department/{departmentId}: Get employees by department ID.
  + **DepartmentController**:
    - POST /api/departments: Create or update a department.
    - GET /api/departments: Get all departments.
    - GET /api/departments/{id}: Get a department by ID.
    - DELETE /api/departments/{id}: Delete a department by ID.

### Exercise 5: Employee Management System - Defining Query Methods

In this exercise, you'll enhance your repositories to support custom queries by defining query methods using keywords and the @Query annotation. You'll also learn how to define and execute named queries using @NamedQuery and @NamedQueries.

### 1. **Defining Query Methods**

#### 1.1 **Using Keywords in Method Names**

Spring Data JPA allows you to create custom query methods by simply defining the method signature with specific keywords.

Let's enhance the EmployeeRepository with a few examples:

1. **Find Employees by First Name**:

java

Copy code

List<Employee> findByFirstName(String firstName);

1. **Find Employees by Last Name Containing a String**:

java

Copy code

List<Employee> findByLastNameContaining(String substring);

1. **Find Employees by Email and Department**:

java

Copy code

Employee findByEmailAndDepartment(String email, Department department);

1. **Find Employees by Department Name**:

java

Copy code

List<Employee> findByDepartmentName(String departmentName);

#### 1.2 **Using the** @Query **Annotation**

For more complex queries, you can use the @Query annotation to define JPQL (Java Persistence Query Language) or native SQL queries.

Let's add custom query methods to the EmployeeRepository:

1. **Find Employees by Email Domain**:

java

Copy code

@Query("SELECT e FROM Employee e WHERE e.email LIKE %:domain")

List<Employee> findByEmailDomain(@Param("domain") String domain);

1. **Find Employees with a Specific Last Name, Ordered by First Name**:

java

Copy code

@Query("SELECT e FROM Employee e WHERE e.lastName = :lastName ORDER BY e.firstName ASC")

List<Employee> findByLastNameOrderedByFirstName(@Param("lastName") String lastName);

### 2. **Named Queries**

Named queries are pre-defined JPQL or SQL queries that can be defined at the entity level. They can be reused across different methods without redefining the query.

#### 2.1 **Define Named Queries in the Entity**

You can define named queries in the Employee entity using @NamedQuery or @NamedQueries.

1. **Define Named Queries in the Employee Entity**:

java

Copy code

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

@Entity

@Table(name = "employees")

@NamedQueries({

@NamedQuery(

name = "Employee.findByDepartmentId",

query = "SELECT e FROM Employee e WHERE e.department.id = :departmentId"

),

@NamedQuery(

name = "Employee.findByEmailDomain",

query = "SELECT e FROM Employee e WHERE e.email LIKE :domain"

)

})

public class Employee {

// Entity fields, constructors, getters, and setters...

}

* **Explanation**:
  + @NamedQueries: A container for multiple @NamedQuery annotations.
  + name: The unique name of the query, which will be used to execute the query.
  + query: The JPQL query string.

#### 2.2 **Execute Named Queries in the Repository**

You can execute these named queries in the repository by using the @Query annotation with the name attribute.

1. **Executing Named Queries in the EmployeeRepository**:

java

Copy code

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

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

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

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Derived query methods...

// Execute named query to find employees by department ID

@Query(name = "Employee.findByDepartmentId")

List<Employee> findEmployeesByDepartmentId(@Param("departmentId") Long departmentId);

// Execute named query to find employees by email domain

@Query(name = "Employee.findByEmailDomain")

List<Employee> findEmployeesByEmailDomain(@Param("domain") String domain);

### Exercise 6: Employee Management System - Implementing Pagination and Sorting

In this exercise, you will add pagination and sorting capabilities to the employee search functionality in your Employee Management System.

### 1. **Implementing Pagination**

Pagination allows you to divide your query results into manageable chunks or pages, which can be very useful when dealing with large datasets.

#### 1.1 **Add Pagination to** EmployeeRepository

Spring Data JPA provides Page and Pageable interfaces to support pagination.

1. **Update EmployeeRepository** to include a paginated query method:

java

Copy code

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

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

import org.springframework.stereotype.Repository;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Other query methods...

// Paginated method to find all employees

Page<Employee> findAll(Pageable pageable);

// Paginated and filtered method to find employees by department ID

Page<Employee> findByDepartmentId(Long departmentId, Pageable pageable);

}

* **Explanation**:
  + Page<Employee>: Represents a page of employees.
  + Pageable: Encapsulates pagination information (page number, page size, etc.).
  + findAll(Pageable pageable): Retrieves a paginated list of all employees.
  + findByDepartmentId(Long departmentId, Pageable pageable): Retrieves a paginated list of employees filtered by department ID.

#### 1.2 **Implement Pagination in the Controller**

Now, you'll update the EmployeeController to support pagination.

1. **Update the EmployeeController**:

java

Copy code

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.service.EmployeeService;

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

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.http.ResponseEntity;

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

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

// Other CRUD endpoints...

// Paginated endpoint to get all employees

@GetMapping("/paginated")

public ResponseEntity<Page<Employee>> getAllEmployeesPaginated(Pageable pageable) {

Page<Employee> employees = employeeService.getAllEmployeesPaginated(pageable);

return ResponseEntity.ok(employees);

}

// Paginated and filtered endpoint to get employees by department ID

@GetMapping("/department/{departmentId}/paginated")

public ResponseEntity<Page<Employee>> getEmployeesByDepartmentIdPaginated(

@PathVariable Long departmentId, Pageable pageable) {

Page<Employee> employees = employeeService.getEmployeesByDepartmentIdPaginated(departmentId, pageable);

return ResponseEntity.ok(employees);

}

}

1. **Update the EmployeeService** to handle pagination:

java

Copy code

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

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

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.stereotype.Service;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

// Other service methods...

// Paginated method to get all employees

public Page<Employee> getAllEmployeesPaginated(Pageable pageable) {

return employeeRepository.findAll(pageable);

}

// Paginated and filtered method to get employees by department ID

public Page<Employee> getEmployeesByDepartmentIdPaginated(Long departmentId, Pageable pageable) {

return employeeRepository.findByDepartmentId(departmentId, pageable);

}

}

### 2. **Implementing Sorting**

Sorting allows you to order your query results based on specified criteria.

#### 2.1 **Add Sorting to** EmployeeRepository

Spring Data JPA provides the Sort class to support sorting.

1. **Update EmployeeRepository** to include sorting methods:

java

Copy code

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

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;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Other query methods...

// Sort method to find all employees

List<Employee> findAll(Sort sort);

// Paginated and sorted method to find all employees

Page<Employee> findAll(Pageable pageable, Sort sort);

// Paginated, filtered, and sorted method to find employees by department ID

Page<Employee> findByDepartmentId(Long departmentId, Pageable pageable, Sort sort);

}

#### 2.2 **Combine Pagination and Sorting in the Controller**

You can combine pagination and sorting by passing a Pageable object with sorting information to your repository methods.

1. **Update the EmployeeController** to support sorting with pagination:

java

Copy code

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.service.EmployeeService;

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

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.domain.Sort;

import org.springframework.http.ResponseEntity;

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

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

// Other CRUD endpoints...

// Paginated and sorted endpoint to get all employees

@GetMapping("/paginated-sorted")

public ResponseEntity<Page<Employee>> getAllEmployeesPaginatedSorted(

Pageable pageable, Sort sort) {

Page<Employee> employees = employeeService.getAllEmployeesPaginatedSorted(pageable, sort);

return ResponseEntity.ok(employees);

}

// Paginated, filtered, and sorted endpoint to get employees by department ID

@GetMapping("/department/{departmentId}/paginated-sorted")

public ResponseEntity<Page<Employee>> getEmployeesByDepartmentIdPaginatedSorted(

@PathVariable Long departmentId, Pageable pageable, Sort sort) {

Page<Employee> employees = employeeService.getEmployeesByDepartmentIdPaginatedSorted(departmentId, pageable, sort);

return ResponseEntity.ok(employees);

}

}

1. **Update the EmployeeService** to handle pagination and sorting:

java

Copy code

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

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

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.domain.Sort;

import org.springframework.stereotype.Service;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

// Other service methods...

// Paginated and sorted method to get all employees

public Page<Employee> getAllEmployeesPaginatedSorted(Pageable pageable, Sort sort) {

return employeeRepository.findAll(pageable, sort);

}

// Paginated, filtered, and sorted method to get employees by department ID

public Page<Employee> getEmployeesByDepartmentIdPaginatedSorted(Long departmentId, Pageable pageable, Sort sort) {

return employeeRepository.findByDepartmentId(departmentId, pageable, sort);

}

}

### Exercise 7: Employee Management System - Enabling Entity Auditing

In this exercise, you'll implement auditing in your Employee Management System to track the creation and modification of employees and departments. Auditing will allow you to automatically capture information about who created or last modified an entity and when these actions occurred.

### 1. **Enable Auditing in Your Application**

To use Spring Data JPA's auditing features, you need to enable auditing in your Spring Boot application.

#### 1.1 **Add Auditing Dependencies (if not already included)**

If you're not using Spring Boot's spring-boot-starter-data-jpa (which includes the auditing support), make sure to add the following dependency in your pom.xml or build.gradle:

xml

Copy code

<dependency>

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

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

</dependency>

#### 1.2 **Enable Auditing in Your Main Application Class**

To enable JPA auditing, you need to add the @EnableJpaAuditing annotation to your main application class.

java

Copy code

package com.example.employeemanagementsystem;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.data.jpa.repository.config.EnableJpaAuditing;

@SpringBootApplication

@EnableJpaAuditing

public class EmployeeManagementSystemApplication {

public static void main(String[] args) {

SpringApplication.run(EmployeeManagementSystemApplication.class, args);

}

}

### 2. **Implement Auditing in Entities**

Now that auditing is enabled, you'll modify your Employee and Department entities to include auditing fields such as createdBy, createdDate, lastModifiedBy, and lastModifiedDate.

#### 2.1 **Add Auditing Fields to Entities**

1. **Create a Base Entity Class for Auditing**:

To avoid repetitive code, create a base class that contains the common auditing fields. Both Employee and Department entities will extend this base class.

java

Copy code

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

import org.springframework.data.annotation.CreatedDate;

import org.springframework.data.annotation.LastModifiedDate;

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

import java.time.LocalDateTime;

@MappedSuperclass

@EntityListeners(AuditingEntityListener.class)

public abstract class Auditable {

@CreatedDate

@Column(updatable = false)

private LocalDateTime createdDate;

@LastModifiedDate

private LocalDateTime lastModifiedDate;

// Getters and Setters

public LocalDateTime getCreatedDate() {

return createdDate;

}

public void setCreatedDate(LocalDateTime createdDate) {

this.createdDate = createdDate;

}

public LocalDateTime getLastModifiedDate() {

return lastModifiedDate;

}

public void setLastModifiedDate(LocalDateTime lastModifiedDate) {

this.lastModifiedDate = lastModifiedDate;

}

}

1. **Extend the Employee Entity from Auditable**:

java

Copy code

package com.example.employeemanagementsystem.model;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.ManyToOne;

@Entity

public class Employee extends Auditable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

@ManyToOne

private Department department;

// Other fields, getters, setters, and constructors

}

1. **Extend the Department Entity from Auditable**:

java

Copy code

package com.example.employeemanagementsystem.model;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

@Entity

public class Department extends Auditable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

// Other fields, getters, setters, and constructors

}

#### 2.2 **Optional: Track** CreatedBy **and** LastModifiedBy

If your application requires tracking of the user who created or last modified the entity, you can add @CreatedBy and @LastModifiedBy annotations.

1. **Extend the Auditable Class**:

java

Copy code

package com.example.employeemanagementsystem.model;

import org.springframework.data.annotation.CreatedBy;

import org.springframework.data.annotation.LastModifiedBy;

import jakarta.persistence.Column;

@MappedSuperclass

@EntityListeners(AuditingEntityListener.class)

public abstract class Auditable {

@CreatedDate

@Column(updatable = false)

private LocalDateTime createdDate;

@LastModifiedDate

private LocalDateTime lastModifiedDate;

@CreatedBy

@Column(updatable = false)

private String createdBy;

@LastModifiedBy

private String lastModifiedBy;

// Getters and Setters

// ...

}

1. **Configure an AuditorAware Implementation**:

You need to tell Spring Security how to fetch the currently authenticated user.

java

Copy code

package com.example.employeemanagementsystem.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.data.domain.AuditorAware;

import java.util.Optional;

@Configuration

public class AuditorAwareImpl implements AuditorAware<String> {

@Override

public Optional<String> getCurrentAuditor() {

// Return the username of the authenticated user

return Optional.of("system"); // Replace "system" with the actual logic to fetch the current user

}

}

1. **Register the AuditorAware Bean**:

java

Copy code

package com.example.employeemanagementsystem.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.data.domain.AuditorAware;

import org.springframework.data.jpa.repository.config.EnableJpaAuditing;

@Configuration

@EnableJpaAuditing(auditorAwareRef = "auditorAware")

public class JpaConfig {

@Bean

public AuditorAware<String> auditorAware() {

return new AuditorAwareImpl();

}

}

### Exercise 8: Employee Management System - Creating Projections

In this exercise, you'll create projections to fetch specific data subsets from the Employee and Department entities. Projections allow you to retrieve only the fields you need, reducing data transfer and improving performance.

### 1. **Understanding Projections**

Projections in Spring Data JPA come in two types:

* **Interface-based Projections**: You define an interface with getter methods corresponding to the fields you want to fetch.
* **Class-based Projections**: You define a class with a constructor that takes the fields you want to fetch.

### 2. **Interface-Based Projections**

#### 2.1 **Define an Interface-Based Projection for Employee**

1. **Create the Projection Interface**:

java

Copy code

package com.example.employeemanagementsystem.projections;

public interface EmployeeNameProjection {

String getName();

String getEmail();

String getDepartmentName();

}

* **Explanation**:
  + The interface defines the fields you want to fetch: name, email, and the department name.
  + The method getDepartmentName() assumes that you will use a join to fetch the department's name.

1. **Modify the EmployeeRepository to Use the Projection**:

java

Copy code

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.projections.EmployeeNameProjection;

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

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

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Other query methods...

@Query("SELECT e.name AS name, e.email AS email, d.name AS departmentName " +

"FROM Employee e JOIN e.department d")

List<EmployeeNameProjection> findEmployeeNamesWithDepartment();

}

* **Explanation**:
  + The findEmployeeNamesWithDepartment() method uses the EmployeeNameProjection interface to return only the required fields.

#### 2.2 **Use the Projection in the Controller**

1. **Modify the EmployeeController to Return the Projection**:

java

Copy code

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.projections.EmployeeNameProjection;

import com.example.employeemanagementsystem.service.EmployeeService;

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

import org.springframework.http.ResponseEntity;

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

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

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

import java.util.List;

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

// Other CRUD endpoints...

@GetMapping("/names")

public ResponseEntity<List<EmployeeNameProjection>> getEmployeeNamesWithDepartment() {

List<EmployeeNameProjection> employeeNames = employeeService.getEmployeeNamesWithDepartment();

return ResponseEntity.ok(employeeNames);

}

}

1. **Modify the EmployeeService to Use the Projection**:

java

Copy code

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.projections.EmployeeNameProjection;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

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

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

// Other service methods...

public List<EmployeeNameProjection> getEmployeeNamesWithDepartment() {

return employeeRepository.findEmployeeNamesWithDepartment();

}

}

### 3. **Class-Based Projections**

#### 3.1 **Define a Class-Based Projection for Department**

1. **Create the Projection Class**:

java

Copy code

package com.example.employeemanagementsystem.projections;

public class DepartmentSummary {

private final String name;

private final long employeeCount;

public DepartmentSummary(String name, long employeeCount) {

this.name = name;

this.employeeCount = employeeCount;

}

// Getters

public String getName() {

return name;

}

public long getEmployeeCount() {

return employeeCount;

}

}

* **Explanation**:
  + The DepartmentSummary class defines the fields name and employeeCount.
  + It includes a constructor that takes these fields as arguments.

1. **Modify the DepartmentRepository to Use the Projection**:

java

Copy code

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Department;

import com.example.employeemanagementsystem.projections.DepartmentSummary;

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

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

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface DepartmentRepository extends JpaRepository<Department, Long> {

// Other query methods...

@Query("SELECT new com.example.employeemanagementsystem.projections.DepartmentSummary(d.name, COUNT(e)) " +

"FROM Department d LEFT JOIN d.employees e " +

"GROUP BY d.name")

List<DepartmentSummary> findDepartmentSummaries();

}

* **Explanation**:
  + The findDepartmentSummaries() method uses the DepartmentSummary class to return the department name and the number of employees.

#### 3.2 **Use the Projection in the Controller**

1. **Modify the DepartmentController to Return the Projection**:

java

Copy code

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.projections.DepartmentSummary;

import com.example.employeemanagementsystem.service.DepartmentService;

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

import org.springframework.http.ResponseEntity;

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

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

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

import java.util.List;

@RestController

@RequestMapping("/api/departments")

public class DepartmentController {

@Autowired

private DepartmentService departmentService;

// Other CRUD endpoints...

@GetMapping("/summaries")

public ResponseEntity<List<DepartmentSummary>> getDepartmentSummaries() {

List<DepartmentSummary> departmentSummaries = departmentService.getDepartmentSummaries();

return ResponseEntity.ok(departmentSummaries);

}

}

1. **Modify the DepartmentService to Use the Projection**:

java

Copy code

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.projections.DepartmentSummary;

import com.example.employeemanagementsystem.repository.DepartmentRepository;

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

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class DepartmentService {

@Autowired

private DepartmentRepository departmentRepository;

// Other service methods...

public List<DepartmentSummary> getDepartmentSummaries() {

return departmentRepository.findDepartmentSummaries();

}

}

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

In class-based projections, you can use the @Value annotation or constructor expressions to further control the fetched data.

#### 4.1 **Use** @Value **for Custom Expressions**

java

Copy code

package com.example.employeemanagementsystem.projections;

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

public class DepartmentSummary {

private final String name;

@Value("#{target.employeeCount + ' employees'}")

private final String employeeCountWithLabel;

public DepartmentSummary(String name, long employeeCount) {

this.name = name;

this.employeeCountWithLabel = employeeCount + " employees";

}

// Getters

public String getName() {

return name;

}

public String getEmployeeCountWithLabel() {

return employeeCountWithLabel;

}

}

### Exercise 9: Employee Management System - Customizing Data Source Configuration

In this exercise, you'll learn how to customize the data source configuration in your Spring Boot Employee Management System, including managing multiple data sources. This is crucial for applications that require different databases for different modules or environments.

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

Spring Boot provides auto-configuration for data sources, which makes it easy to get started with a database connection. By default, Spring Boot will configure a data source if it detects spring.datasource properties in the application.properties or application.yml file.

#### 1.1 **Basic Auto-Configuration**

If you're using a single data source, you can rely on Spring Boot's auto-configuration by defining the data source properties in application.properties:

properties

Copy code

# application.properties

# Basic Data Source Configuration

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

spring.datasource.username=sa

spring.datasource.password=password

spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

* **Explanation**:
  + The spring.datasource.url defines the database URL.
  + The spring.datasource.username and spring.datasource.password set the credentials.
  + spring.datasource.driver-class-name specifies the JDBC driver.
  + spring.jpa.database-platform configures the JPA dialect for the database.

### 2. **Externalizing Configuration**

Externalizing configuration allows you to manage properties in different environments without modifying your code. Spring Boot supports externalized configuration through application.properties or application.yml, environment variables, and command-line arguments.

#### 2.1 **Using** application.properties **for Externalized Configuration**

You can define data source properties in an application.properties file that resides in your src/main/resources directory:

properties

Copy code

# application.properties

# Development Database

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

spring.datasource.username=devuser

spring.datasource.password=devpass

spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Production Database

# You can override these properties in an application-prod.properties file

# spring.datasource.url=jdbc:mysql://localhost/proddb

# spring.datasource.username=produser

# spring.datasource.password=prodpass

# spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect

* **Explanation**:
  + This setup uses different profiles for different environments. For instance, the application-prod.properties file can contain production-specific settings that override the default ones.

#### 2.2 **Activating Profiles**

You can activate a specific profile (e.g., prod) by passing the profile as a command-line argument:

bash

Copy code

java -jar EmployeeManagementSystem.jar --spring.profiles.active=prod

Or by setting it in your IDE's run configuration.

### 3. **Managing Multiple Data Sources**

In some scenarios, your application might need to connect to multiple databases. Spring Boot allows you to configure multiple data sources by defining custom configuration classes.

#### 3.1 **Configuring Multiple Data Sources**

1. **Define Data Source Properties in application.properties**:

properties

Copy code

# application.properties

# First Data Source (e.g., Primary Database)

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

spring.datasource.primary.username=primaryuser

spring.datasource.primary.password=primarypass

spring.datasource.primary.driver-class-name=org.h2.Driver

# Second Data Source (e.g., Secondary Database)

spring.datasource.secondary.url=jdbc:h2:mem:secondarydb

spring.datasource.secondary.username=secondaryuser

spring.datasource.secondary.password=secondarypass

spring.datasource.secondary.driver-class-name=org.h2.Driver

1. **Create Configuration Classes for Each Data Source**:

java

Copy code

package com.example.employeemanagementsystem.config;

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

import org.springframework.boot.autoconfigure.orm.jpa.JpaProperties;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.jdbc.DataSourceBuilder;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

import org.springframework.orm.jpa.JpaTransactionManager;

import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;

import org.springframework.transaction.PlatformTransactionManager;

import javax.sql.DataSource;

import java.util.HashMap;

@Configuration

public class DataSourceConfig {

@Primary

@Bean(name = "primaryDataSource")

@ConfigurationProperties(prefix = "spring.datasource.primary")

public DataSource primaryDataSource() {

return DataSourceBuilder.create().build();

}

@Bean(name = "secondaryDataSource")

@ConfigurationProperties(prefix = "spring.datasource.secondary")

public DataSource secondaryDataSource() {

return DataSourceBuilder.create().build();

}

@Primary

@Bean(name = "primaryEntityManagerFactory")

public LocalContainerEntityManagerFactoryBean primaryEntityManagerFactory(

@Qualifier("primaryDataSource") DataSource primaryDataSource, JpaProperties jpaProperties) {

LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();

em.setDataSource(primaryDataSource);

em.setPackagesToScan("com.example.employeemanagementsystem.model.primary");

em.setJpaVendorAdapter(new HibernateJpaVendorAdapter());

em.setJpaPropertyMap(new HashMap<>(jpaProperties.getProperties()));

return em;

}

@Bean(name = "secondaryEntityManagerFactory")

public LocalContainerEntityManagerFactoryBean secondaryEntityManagerFactory(

@Qualifier("secondaryDataSource") DataSource secondaryDataSource, JpaProperties jpaProperties) {

LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();

em.setDataSource(secondaryDataSource);

em.setPackagesToScan("com.example.employeemanagementsystem.model.secondary");

em.setJpaVendorAdapter(new HibernateJpaVendorAdapter());

em.setJpaPropertyMap(new HashMap<>(jpaProperties.getProperties()));

return em;

}

@Primary

@Bean(name = "primaryTransactionManager")

public PlatformTransactionManager primaryTransactionManager(

@Qualifier("primaryEntityManagerFactory") LocalContainerEntityManagerFactoryBean primaryEntityManagerFactory) {

return new JpaTransactionManager(primaryEntityManagerFactory.getObject());

}

@Bean(name = "secondaryTransactionManager")

public PlatformTransactionManager secondaryTransactionManager(

@Qualifier("secondaryEntityManagerFactory") LocalContainerEntityManagerFactoryBean secondaryEntityManagerFactory) {

return new JpaTransactionManager(secondaryEntityManagerFactory.getObject());

}

}

* **Explanation**:
  + Each data source has its own configuration, including entity manager factory and transaction manager.
  + The @Primary annotation indicates which data source is the default one.

1. **Creating Repositories for Each Data Source**:

You can specify which entity manager to use for each repository by annotating the repository interfaces with the @Primary qualifier.

java

Copy code

package com.example.employeemanagementsystem.repository.primary;

import com.example.employeemanagementsystem.model.primary.Employee;

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

import org.springframework.stereotype.Repository;

@Repository

public interface PrimaryEmployeeRepository extends JpaRepository<Employee, Long> {

// Query methods

}

java

Copy code

package com.example.employeemanagementsystem.repository.secondary;

import com.example.employeemanagementsystem.model.secondary.Department;

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

import org.springframework.stereotype.Repository;

@Repository

public interface SecondaryDepartmentRepository extends JpaRepository<Department, Long> {

// Query methods

}

* **Explanation**:
  + The PrimaryEmployeeRepository uses the primary data source.
  + The SecondaryDepartmentRepository uses the secondary data source.

### Exercise 10: Employee Management System - Hibernate-Specific Features

In this exercise, you will leverage Hibernate-specific features to enhance the performance and capabilities of your Employee Management System. Hibernate offers several advanced features for optimizing entity mappings, configuring performance settings, and handling batch processing.

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

Hibernate provides additional annotations beyond JPA standard annotations that you can use to customize your entity mappings.

#### 1.1 **Using Hibernate Annotations**

1. **@Type Annotation**:

This annotation allows you to specify a custom Hibernate type for an attribute.

java

Copy code

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@Type(type = "string")

private String customField;

// Other fields, getters, setters

}

1. **@BatchSize Annotation**:

This annotation is used to define batch sizes for fetching collections, which can help improve performance by reducing the number of queries.

java

Copy code

@Entity

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "department")

@BatchSize(size = 20) // Adjust the batch size as needed

private List<Employee> employees;

// Other fields, getters, setters

}

1. **@Where Annotation**:

This annotation allows you to add a filter to your entity to fetch only the records that meet specific criteria.

java

Copy code

@Entity

@Where(clause = "active = true")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private boolean active;

// Other fields, getters, setters

}

* + **Explanation**:
    - The @Where annotation filters out inactive employees by default.

1. **@Fetch Annotation**:

This annotation controls how associations are fetched.

java

Copy code

@Entity

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "department", fetch = FetchType.LAZY)

@Fetch(FetchMode.SUBSELECT) // Use subselect to optimize loading

private List<Employee> employees;

// Other fields, getters, setters

}

* + **Explanation**:
    - FetchMode.SUBSELECT loads collections in a single query for performance optimization.

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

To optimize Hibernate performance, configure the Hibernate dialect and properties in application.properties or application.yml.

#### 2.1 **Configuring Hibernate Dialect**

The dialect determines the SQL syntax Hibernate uses for your database. Specify it in your configuration:

properties

Copy code

# application.properties

# Hibernate Dialect

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect

* **Explanation**:
  + Replace H2Dialect with the appropriate dialect for your database (e.g., MySQLDialect for MySQL).

#### 2.2 **Configuring Hibernate Properties**

Configure additional Hibernate properties for performance tuning:

properties

Copy code

# application.properties

# Hibernate Properties

spring.jpa.properties.hibernate.show\_sql=true

spring.jpa.properties.hibernate.format\_sql=true

spring.jpa.properties.hibernate.use\_sql\_comments=true

spring.jpa.properties.hibernate.jdbc.batch\_size=30

spring.jpa.properties.hibernate.order\_inserts=true

spring.jpa.properties.hibernate.order\_updates=true

* **Explanation**:
  + hibernate.show\_sql prints SQL statements to the console.
  + hibernate.format\_sql formats SQL statements for readability.
  + hibernate.use\_sql\_comments adds comments to SQL statements.
  + hibernate.jdbc.batch\_size sets the batch size for batch processing.
  + hibernate.order\_inserts and hibernate.order\_updates optimize the order of SQL statements for batch processing.

### 3. **Batch Processing**

Batch processing is useful for handling bulk operations efficiently.

#### 3.1 **Enabling Batch Processing**

Configure Hibernate to support batch processing in application.properties:

properties

Copy code

# application.properties

# Hibernate Batch Processing Configuration

spring.jpa.properties.hibernate.jdbc.batch\_size=50

spring.jpa.properties.hibernate.order\_inserts=true

spring.jpa.properties.hibernate.order\_updates=true

* **Explanation**:
  + hibernate.jdbc.batch\_size specifies the number of inserts/updates to batch in a single round-trip to the database.

#### 3.2 **Using Batch Processing in Your Application**

Batch processing can be implemented using the repository or service layer. Here's an example of bulk inserting employees:

java

Copy code

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void saveEmployeesInBatch(List<Employee> employees) {

final int batchSize = 50; // Adjust the batch size as needed

for (int i = 0; i < employees.size(); i++) {

employeeRepository.save(employees.get(i));

if (i % batchSize == 0 && i > 0) {

// Flush a batch of inserts and release memory

employeeRepository.flush();

employeeRepository.clear();

}

}

}

}

* **Explanation**:
  + @Transactional ensures that the operation is performed within a transaction.
  + The flush() method is used to synchronize the persistence context with the database, and clear() releases memory.