Exercise 1: Employee Management System - Overview and Setup

1.

- **Project Name:** EmployeeManagementSystem
- **Group:** com.example
- **Artifact:** EmployeeManagementSystem
- **Packaging:** jar
- **Java Version:** 17 (or any supported version)
- **Dependencies:**
  - **Spring Data JPA:** For managing database operations.
  - **H2 Database:** An in-memory database for quick testing and development.
  - **Spring Web:** For creating RESTful APIs.
  - **Lombok:** To reduce boilerplate code like getters, setters, constructors, etc.

2. **Generate the project** and download it. Extract the project and open it in your IDE (IntelliJ IDEA or Eclipse).

Step 2: Configuring Application Properties

Once your project is set up, you need to configure the application.properties file to connect to the H2 database.

1. **Locate the application.properties file:**

This file is located in src/main/resources/application.properties. If it's not present, create one.

```
# H2 Database Configuration
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password

# Hibernate Configuration for H2
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Enable H2 Console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

- **spring.datasource.url=jdbc:h2:mem**

: Configures the H2 database in memory mode. testdb is the name of the database.

- **spring.datasource.driverClassName=org.h2.Driver**: Specifies the driver class for H2 database.
- **spring.datasource.username=sa**: Username for the database.
- **spring.datasource.password=password**: Password for the database.
- **spring.jpa.database-platform=org.hibernate.dialect.H2Dialect**: Configures the dialect for H2 database with Hibernate.
- **spring.h2.console.enabled=true**: Enables the H2 database console, which can be accessed via a web browser.
- **spring.h2.console.path=/h2-console**: Specifies the path to access the H2 console (e.g., http://localhost:8080/h2-console).

Final Steps:

1. **Run the Application:**
    - Navigate to the EmployeeManagementSystemApplication.java file and run it as a Spring Boot application.
    - This will start the embedded server (Tomcat by default) and configure the H2 database.
2. **Access the H2 Console:**
    - Open a web browser and go to http://localhost:8080/h2-console.
    - Use the configured settings to connect:
        - **JDBC URL:** jdbc:h2:mem:testdb
        - **Username:** sa
        - **Password:** password
    - Click "Connect," and you can now interact with the H2 database.

Exercise 2: Employee Management System - Creating Entities

Business Scenario:

Define JPA entities for Employee and Department with appropriate relationships.
Employee Entity:

```
package com.example.employeemanagementsystem.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Table(name = "employees")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Employee {

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

    private String name;
    private String email;

    @ManyToOne
```

```
    @JoinColumn(name = "department_id", nullable = false)
    private Department department;
}
```

Department Entity:

```
package com.example.employeemanagementsystem.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToMany;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.List;

@Entity
@Table(name = "departments")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Department {

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

    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees;
}
```

Mapping Entities to Database Tables

- **Annotations:**
    - @Entity: Marks the class as a JPA entity.
    - @Table: Specifies the table name in the database (optional, defaults to the class name).
    - @Id: Marks the primary key field.
    - @GeneratedValue: Configures the strategy for primary key generation (e.g., IDENTITY).
    - @ManyToOne: Defines a many-to-one relationship from Employee to Department.
    - @OneToMany: Defines a one-to-many relationship from Department to Employee.
    - @JoinColumn: Specifies the foreign key column in the Employee table.

Explanation of the Relationship

- **Employee and Department:**
  - Each Employee belongs to a single Department, represented by a ManyToOne relationship.
  - A Department can have multiple Employees, represented by a OneToMany relationship.

Final Steps

After defining these entities:

1. **Create Repositories:** Create Spring Data JPA repositories for Employee and Department to handle database operations.
2. **Create Services and Controllers:** Implement services and controllers to manage business logic and expose RESTful APIs.

Your entities are now ready, and you can proceed with further implementation to manage the employee and department data.

Exercise 3: Employee Management System - Creating Repositories

Create EmployeeRepository Interface:

```
package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.entity.Employee;
import org.springframework.data.jpa.repository.JpaRepository;
import java.util.List;

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 employees by name
    List<Employee> findByName(String name);
}
```

Create DepartmentRepository Interface:

```
package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.entity.Department;
import org.springframework.data.jpa.repository.JpaRepository;

public interface DepartmentRepository extends JpaRepository<Department, Long> {

    // Derived query method to find departments by name
    Department findByName(String name);
}
```

- **JpaRepository<T, ID>**: This interface provides JPA-related methods for standard CRUD operations. T is the entity type, and ID is the type of the entity's primary key.
- **Derived Query Methods:** Spring Data JPA allows you to define methods in the repository interface that follow a certain naming convention, like findBy, countBy, deleteBy, etc. Spring Data will automatically generate the necessary query based on the method name.

Final Steps

1. **Inject Repositories into Services:** You can now inject these repositories into your service classes to perform business logic operations.
2. **Testing:** Test the repositories by creating, reading, updating, and deleting entities via your service layer or directly in a test class.
3. **Extend Functionality:** Add more custom queries or business logic as needed.

With these repositories in place, you can now efficiently perform CRUD operations on Employee and Department entities in your Employee Management System.

Exercise 4: Employee Management System - Implementing CRUD Operations

EmployeeService

```
package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.entity.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 Employee
    public Employee saveEmployee(Employee employee) {
        return employeeRepository.save(employee);
    }

    // Read Employee by ID
    public Optional<Employee> getEmployeeById(Long id) {
        return employeeRepository.findById(id);
    }

    // Read All Employees
    public List<Employee> getAllEmployees() {
        return employeeRepository.findAll();
```

```
    }

    // Delete Employee by ID
    public void deleteEmployee(Long id) {
        employeeRepository.deleteById(id);
    }
}
```

DepartmentService

```
package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.entity.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 Department
    public Department saveDepartment(Department department) {
        return departmentRepository.save(department);
    }

    // Read Department by ID
    public Optional<Department> getDepartmentById(Long id) {
        return departmentRepository.findById(id);
    }

    // Read All Departments
    public List<Department> getAllDepartments() {
        return departmentRepository.findAll();
    }

    // Delete Department by ID
    public void deleteDepartment(Long id) {
        departmentRepository.deleteById(id);
    }
}
```

Implementing RESTful Endpoints in Controllers

EmployeeController

```
package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.entity.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;
import java.util.Optional;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    // Create a new Employee
    @PostMapping
    public Employee createEmployee(@RequestBody Employee employee) {
        return employeeService.saveEmployee(employee);
    }

    // Get an Employee by ID
    @GetMapping("/{id}")
    public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id) {
        Optional<Employee> employee = employeeService.getEmployeeById(id);
        return employee.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());
    }

    // Get all Employees
    @GetMapping
    public List<Employee> getAllEmployees() {
        return employeeService.getAllEmployees();
    }

    // Update an Employee
    @PutMapping("/{id}")
    public ResponseEntity<Employee> updateEmployee(@PathVariable Long id, @RequestBody
Employee employeeDetails) {
        Optional<Employee> employee = employeeService.getEmployeeById(id);
        if (employee.isPresent()) {
            Employee existingEmployee = employee.get();
            existingEmployee.setName(employeeDetails.getName());
            existingEmployee.setEmail(employeeDetails.getEmail());
            existingEmployee.setDepartment(employeeDetails.getDepartment());
            Employee updatedEmployee = employeeService.saveEmployee(existingEmployee);
```

```
            return ResponseEntity.ok(updatedEmployee);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Delete an Employee
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteEmployee(@PathVariable Long id) {
        Optional<Employee> employee = employeeService.getEmployeeById(id);
        if (employee.isPresent()) {
            employeeService.deleteEmployee(id);
            return ResponseEntity.noContent().build();
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```

DepartmentController

```
package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.entity.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;
import java.util.Optional;

@RestController
@RequestMapping("/api/departments")
public class DepartmentController {

    @Autowired
    private DepartmentService departmentService;

    // Create a new Department
    @PostMapping
    public Department createDepartment(@RequestBody Department department) {
        return departmentService.saveDepartment(department);
    }

    // Get a Department by ID
    @GetMapping("/{id}")
    public ResponseEntity<Department> getDepartmentById(@PathVariable Long id) {
        Optional<Department> department = departmentService.getDepartmentById(id);
```

```
        return department.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());
    }

    // Get all Departments
    @GetMapping
    public List<Department> getAllDepartments() {
        return departmentService.getAllDepartments();
    }

    // Update a Department
    @PutMapping("/{id}")
    public ResponseEntity<Department> updateDepartment(@PathVariable Long id, @RequestBody
Department departmentDetails) {
        Optional<Department> department = departmentService.getDepartmentById(id);
        if (department.isPresent()) {
            Department existingDepartment = department.get();
            existingDepartment.setName(departmentDetails.getName());
            Department updatedDepartment = departmentService.saveDepartment(existingDepartment);
            return ResponseEntity.ok(updatedDepartment);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Delete a Department
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteDepartment(@PathVariable Long id) {
        Optional<Department> department = departmentService.getDepartmentById(id);
        if (department.isPresent()) {
            departmentService.deleteDepartment(id);
            return ResponseEntity.noContent().build();
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```

- **RESTful Endpoints**:
    - @PostMapping: To create a new entity (Employee or Department).
    - @GetMapping: To retrieve entities by ID or list all entities.
    - @PutMapping: To update an existing entity.
    - @DeleteMapping: To delete an entity by ID.
- **ResponseEntity**: Provides a standardized way to return HTTP responses, including status codes (e.g., 200 OK, 404 Not Found, 204 No Content).

Final Steps

1. **Test the API Endpoints**:
    - Use tools like Postman or curl to send HTTP requests to the API endpoints.
    - Ensure all CRUD operations work as expected.

2. **Expand Functionality**:
    - Implement additional business logic as needed.
    - Add validation, exception handling, and more complex queries or services.

Your Employee Management System now supports full CRUD operations via RESTful endpoints for both employees and departments.

Exercise 5: Employee Management System - Defining Query Methods

EmployeeRepository

```
package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.entity.Employee;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    // Find employees by their name
    List<Employee> findByName(String name);

    // Find employees by their email
    Employee findByEmail(String email);

    // Find employees by their department's name
    List<Employee> findByDepartmentName(String departmentName);

    // Find employees by their department ID and name
    List<Employee> findByDepartmentIdAndName(Long departmentId, String name);

    // Find employees whose name contains a specific string
    List<Employee> findByNameContaining(String keyword);
}
```

DepartmentRepository

```
package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.entity.Department;
import org.springframework.data.jpa.repository.JpaRepository;

public interface DepartmentRepository extends JpaRepository<Department, Long> {

    // Find a department by its name
```

```
    Department findByName(String name);

    // Find departments whose name starts with a specific prefix
    List<Department> findByNameStartingWith(String prefix);

    // Find departments whose name ends with a specific suffix
    List<Department> findByNameEndingWith(String suffix);

    // Find departments whose name contains a specific string
    List<Department> findByNameContaining(String keyword);
}
```

Implementing Custom Query Methods Using the @Query Annotation

EmployeeRepository

```
package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.entity.Employee;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    // Custom query to find employees by department's name
    @Query("SELECT e FROM Employee e WHERE e.department.name = :departmentName")
    List<Employee> findEmployeesByDepartmentName(@Param("departmentName") String
departmentName);

    // Custom query to find employees by partial email match
    @Query("SELECT e FROM Employee e WHERE e.email LIKE %:emailFragment%")
    List<Employee> findEmployeesByEmailFragment(@Param("emailFragment") String
emailFragment);
}
```

DepartmentRepository

```
package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.entity.Department;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

public interface DepartmentRepository extends JpaRepository<Department, Long> {
```

```
    // Custom query to find departments with a specific number of employees
    @Query("SELECT d FROM Department d WHERE SIZE(d.employees) = :employeeCount")
    List<Department> findDepartmentsByEmployeeCount(@Param("employeeCount") int
employeeCount);
}
```

Named Queries

Named queries are pre-defined, reusable queries that can be declared at the entity level. They are typically used for more complex or frequently used queries.

Employee Entity with Named Query

```
package com.example.employeemanagementsystem.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.NamedQueries;
import jakarta.persistence.NamedQuery;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Table(name = "employees")
@Data
@NoArgsConstructor
@AllArgsConstructor
@NamedQueries({
    @NamedQuery(name = "Employee.findByDepartmentName",
            query = "SELECT e FROM Employee e WHERE e.department.name = :departmentName"),
    @NamedQuery(name = "Employee.findByEmailFragment",
            query = "SELECT e FROM Employee e WHERE e.email LIKE :emailFragment")
})
public class Employee {

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

    private String name;
    private String email;

    @ManyToOne
    private Department department;
```

```
}
```

Use Named Queries in Repository

```
package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.entity.Employee;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.repository.query.Param;

import java.util.List;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    // Named query method to find employees by department name
    List<Employee> findByDepartmentName(@Param("departmentName") String departmentName);

    // Named query method to find employees by partial email match
    List<Employee> findByEmailFragment(@Param("emailFragment") String emailFragment);
}
```

Explanation of Named Queries

- **@NamedQuery**: Defines a single named query that can be reused in the repository. The name attribute is the identifier, and the query attribute is the JPQL query.
- **@NamedQueries**: Container for multiple @NamedQuery annotations.
- **Usage**: You can call named queries directly from your repository interface using their names.

Final Steps

1. **Test Custom Queries**: Use integration tests or tools like Postman to verify that your custom queries work as expected.
2. **Optimize Queries**: Review and optimize queries for performance, particularly if you're working with large datasets.
3. **Expand Query Methods**: Add more query methods to handle specific business logic or reporting needs as your application grows.

With these custom query methods and named queries in place, your Employee Management System is now more flexible and powerful, allowing you to handle complex data retrieval scenarios.

Exercise 6: Employee Management System - Implementing Pagination and Sorting

To implement pagination and sorting in an Employee Management System, you typically use Spring Data JPA if you're working with a Java-based backend. Here's a step-by-step guide to help you implement these features.

1. Setup and Dependencies

Ensure that you have the following dependencies in your pom.xml (if using Maven):

```
<dependencies>
    <!-- Other dependencies -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

2. Entity Class

Assume you have an Employee entity:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String department;
    private double salary;
    // Getters and Setters
}
```

3. Repository Interface

Your repository interface should extend JpaRepository, which provides methods for pagination and sorting:

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

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
}
```

4. Service Layer

In the service layer, create a method to fetch paginated and sorted results:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
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;
```

```
    public Page<Employee> getAllEmployees(int page, int size, String sortBy, String sortDir) {
        Sort sort = sortDir.equalsIgnoreCase(Sort.Direction.ASC.name()) ? Sort.by(sortBy).ascending()
                : Sort.by(sortBy).descending();

        Pageable pageable = PageRequest.of(page, size, sort);
        return employeeRepository.findAll(pageable);
    }
}
```

5. Controller Layer

Expose the service in a REST controller:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/employees")
    public Page<Employee> getEmployees(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(defaultValue = "id") String sortBy,
            @RequestParam(defaultValue = "asc") String sortDir) {

        return employeeService.getAllEmployees(page, size, sortBy, sortDir);
    }
}
```

6. Testing the Endpoint

You can test the endpoint by hitting the following URL:

http://localhost:8080/employees?page=0&size=5&sortBy=name&sortDir=desc

This will return the first page of employees, with 5 employees per page, sorted by name in descending order.

7. Handling Edge Cases (Optional)

- **Validation:** Validate the page, size, sortBy, and sortDir parameters to ensure they are within acceptable ranges/values.
- **Error Handling:** Implement error handling to manage scenarios like invalid sorting fields or directions.

8. Front-End Considerations

If you're working with a front-end, you'll typically display pagination controls (like "Next", "Previous", and page numbers) and allow the user to choose sorting options.

Exercise 7: Employee Management System - Enabling Entity Auditing

To enable entity auditing in your Employee Management System using Spring Data JPA, you can follow these steps:

1. Add Dependencies

Ensure your pom.xml has the necessary dependencies:

```
<dependencies>
  <!-- Other dependencies -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
</dependencies>
```

2. Enable Auditing in the Application

You need to enable auditing in your Spring Boot application. To do this, annotate your main application class with @EnableJpaAuditing.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.jpa.repository.config.EnableJpaAuditing;

@SpringBootApplication
@EnableJpaAuditing
public class EmployeeManagementApplication {
  public static void main(String[] args) {
    SpringApplication.run(EmployeeManagementApplication.class, args);
  }
}
```

3. Create Auditable Entity Base Class

Create a base class that other entities can inherit from. This class will include common auditing fields.

```
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import javax.persistence.EntityListeners;
import javax.persistence.MappedSuperclass;
import java.time.LocalDateTime;

@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class Auditable {

  @CreatedDate
  private LocalDateTime createdDate;
```

```
    @LastModifiedDate
    private LocalDateTime lastModifiedDate;

    // Getters and Setters
}
```

4. Add Auditing Fields to Entities

Now, modify your Employee and Department entities to extend the Auditable base class and add additional fields like @CreatedBy and @LastModifiedBy.

For Employee:

```
import org.springframework.data.annotation.CreatedBy;
import org.springframework.data.annotation.LastModifiedBy;

import javax.persistence.Entity;

@Entity
public class Employee extends Auditable {

    private String name;
    private String department;
    private double salary;

    @CreatedBy
    private String createdBy;

    @LastModifiedBy
    private String lastModifiedBy;

    // Getters and Setters
}
```

For Department:

```
import org.springframework.data.annotation.CreatedBy;
import org.springframework.data.annotation.LastModifiedBy;

import javax.persistence.Entity;

@Entity
public class Department extends Auditable {

    private String name;

    @CreatedBy
    private String createdBy;
    @LastModifiedBy
    private String lastModifiedBy;

    // Getters and Setters
```

```
}
```

5. Configure AuditorAware Implementation

You need to implement the AuditorAware interface to specify the current user or system performing the operation.

```
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() {
      // Here you can fetch the current logged-in user, for now returning a fixed value
      return Optional.of("admin"); // Replace with the actual user fetching logic
   }
}
```

6. Register AuditorAware Bean

Register the AuditorAware implementation as a bean in your configuration.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.domain.AuditorAware;

@Configuration
public class AuditConfig {

   @Bean
   public AuditorAware<String> auditorProvider() {
      return new AuditorAwareImpl();
   }
}
```

7. Database Changes

Make sure your database schema supports these auditing fields. If you are using an ORM tool like Hibernate, the schema will be updated automatically if hibernate.ddl-auto is set to update or create.

8. Testing the Auditing

- Create or Update Entities: When you create or modify an Employee or Department, the fields createdDate, lastModifiedDate, createdBy, and lastModifiedBy should be automatically populated.
- Verify Data: Check the database to ensure that these fields are being correctly updated.

Exercise 8: Employee Management System – Creating Projections

In the Employee Management System, projections allow you to fetch specific subsets of data from your entities rather than retrieving entire entities. This is especially useful when you want to optimize performance by limiting the amount of data retrieved from the database.

1. Interface-Based Projections

Interface-based projections are a simple way to define a subset of fields you want to retrieve from an entity.

1.1 Define Interface-Based Projection for Employee

Create an interface that defines the fields you want to project.

java

Copy code

```
public interface EmployeeNameAndDepartment {
   String getName();
   String getDepartment();
}
```

1.2 Define Interface-Based Projection for Department

Similarly, define a projection for the Department entity:

java

Copy code

```
public interface DepartmentNameAndId {
   Long getId();
   String getName();
}
```

1.3 Use Projections in Repository

In your repository interface, add methods that return these projections:

java

Copy code

```
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> {

   // Projection with a derived query method
   List<EmployeeNameAndDepartment> findAllByDepartment(String department);

   // Projection using a custom query
   @Query("SELECT e.name AS name, e.department AS department FROM Employee e")
   List<EmployeeNameAndDepartment> findEmployeeNamesAndDepartments();
}

@Repository
public interface DepartmentRepository extends JpaRepository<Department, Long> {
```

```
    List<DepartmentNameAndId> findBy();
}
```

2. Class-Based Projections

Class-based projections use a DTO (Data Transfer Object) class to represent the subset of data. You can use constructor expressions to achieve this.

2.1 Define DTO for Employee

Create a DTO class that will hold the projected data.

java

Copy code

```
public class EmployeeDTO {

    private String name;
    private String department;

    // Constructor for the projection
    public EmployeeDTO(String name, String department) {
        this.name = name;
        this.department = department;
    }

    // Getters and Setters
}
```

2.2 Use DTO in Repository

In the repository interface, create a query that returns the DTO:

java

Copy code

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

import java.util.List;

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    @Query("SELECT new com.example.demo.EmployeeDTO(e.name, e.department) FROM Employee
e")
    List<EmployeeDTO> findEmployeeDTOs();
}
```

3. Using @Value and Constructor Expressions

Sometimes, you might want to create more complex projections or derive values on the fly. You can do this using the @Value annotation and SpEL (Spring Expression Language).

3.1 Create a Projection with Derived Fields

Suppose you want to include a full name (firstName + lastName) in your projection. You can achieve this using @Value.

java

Copy code

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

```
public interface EmployeeFullNameProjection {

    @Value("#{target.firstName + ' ' + target.lastName}")
    String getFullName();

    String getDepartment();
}
```

3.2 Use the Projection in Repository

java

Copy code

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

import java.util.List;

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    List<EmployeeFullNameProjection> findByDepartment(String department);
}
```

4. Fetching and Using Projections

You can fetch and use projections in your service or controller layer like this:

java

Copy code

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    public List<EmployeeNameAndDepartment> getEmployeeNameAndDepartment(String
department) {
        return employeeRepository.findAllByDepartment(department);
    }

    public List<EmployeeDTO> getEmployeeDTOs() {
        return employeeRepository.findEmployeeDTOs();
    }

    public List<EmployeeFullNameProjection> getEmployeeFullNames(String department) {
        return employeeRepository.findByDepartment(department);
    }
}
```

Exercise 9: Employee Management System – Customizing Data Source Configuration

To customize the data source configuration in your Employee Management System and manage multiple data sources, follow these steps:

1. Spring Boot Auto-Configuration

Spring Boot simplifies data source configuration with auto-configuration. By default, Spring Boot can auto-configure a DataSource bean based on the properties defined in application.properties or application.yml.

1.1 Default Data Source Configuration

If you only have one data source, you can configure it in the application.properties file:

properties

Copy code

```
spring.datasource.url=jdbc:mysql://localhost:3306/employee_db
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

With this configuration, Spring Boot automatically creates a DataSource bean and uses it for all database operations.

2. Externalizing Configuration

Externalizing configuration means moving your configurations, such as database credentials, out of your codebase. This is typically done using application.properties or application.yml files.

2.1 Externalize Configuration with application.properties

You can externalize the configuration as shown above in application.properties. Additionally, you can use environment variables or externalized configuration files to override these values at runtime.

For example, you can define placeholders:

properties

Copy code

```
spring.datasource.url=${DB_URL:jdbc:mysql://localhost:3306/employee_db}
spring.datasource.username=${DB_USERNAME:root}
spring.datasource.password=${DB_PASSWORD:yourpassword}
```

You can then provide the actual values via environment variables or an external properties file, allowing different environments (dev, test, prod) to use different configurations.

3. Managing Multiple Data Sources

In some applications, you may need to manage multiple data sources, such as one for employee data and another for department data.

3.1 Define Multiple Data Sources in application.properties

First, define properties for each data source in application.properties:

properties

Copy code

```
# Primary Data Source (Employee)
spring.datasource.employee.url=jdbc:mysql://localhost:3306/employee_db
spring.datasource.employee.username=root
spring.datasource.employee.password=yourpassword
spring.datasource.employee.driver-class-name=com.mysql.cj.jdbc.Driver

# Secondary Data Source (Department)
```

```
spring.datasource.department.url=jdbc:mysql://localhost:3306/department_db
spring.datasource.department.username=root
spring.datasource.department.password=yourpassword
spring.datasource.department.driver-class-name=com.mysql.cj.jdbc.Driver
```

3.2 Create Configuration Classes for Multiple Data Sources

Next, create configuration classes to define the DataSource beans:

java

Copy code

```
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.persistence.EntityManagerFactory;
import javax.sql.DataSource;

@Configuration
@EnableJpaRepositories(
    basePackages = "com.example.employee.repository", // Specify the repository package for
employee
    entityManagerFactoryRef = "employeeEntityManagerFactory",
    transactionManagerRef = "employeeTransactionManager"
)
public class EmployeeDataSourceConfig {

  @Primary
  @Bean(name = "employeeDataSource")
  @ConfigurationProperties(prefix = "spring.datasource.employee")
  public DataSource employeeDataSource() {
    return DataSourceBuilder.create().build();
  }

  @Primary
  @Bean(name = "employeeEntityManagerFactory")
  public LocalContainerEntityManagerFactoryBean employeeEntityManagerFactory() {
    LocalContainerEntityManagerFactoryBean em = new
LocalContainerEntityManagerFactoryBean();
    em.setDataSource(employeeDataSource());
    em.setPackagesToScan(new String[]{"com.example.employee.entity"}); // Entity package

    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    em.setJpaVendorAdapter(vendorAdapter);
```

```
        return em;
    }

    @Primary
    @Bean(name = "employeeTransactionManager")
    public PlatformTransactionManager employeeTransactionManager(EntityManagerFactory
employeeEntityManagerFactory) {
        return new JpaTransactionManager(employeeEntityManagerFactory);
    }
}
```

Then, create a similar configuration for the Department data source:

java

Copy code

```
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.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.persistence.EntityManagerFactory;
import javax.sql.DataSource;

@Configuration
@EnableJpaRepositories(
        basePackages = "com.example.department.repository", // Specify the repository package for
department
        entityManagerFactoryRef = "departmentEntityManagerFactory",
        transactionManagerRef = "departmentTransactionManager"
)
public class DepartmentDataSourceConfig {

    @Bean(name = "departmentDataSource")
    @ConfigurationProperties(prefix = "spring.datasource.department")
    public DataSource departmentDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "departmentEntityManagerFactory")
    public LocalContainerEntityManagerFactoryBean departmentEntityManagerFactory() {
        LocalContainerEntityManagerFactoryBean em = new
LocalContainerEntityManagerFactoryBean();
        em.setDataSource(departmentDataSource());
        em.setPackagesToScan(new String[]{"com.example.department.entity"}); // Entity package
```

```
        HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        em.setJpaVendorAdapter(vendorAdapter);

        return em;
    }

    @Bean(name = "departmentTransactionManager")
    public PlatformTransactionManager departmentTransactionManager(EntityManagerFactory
departmentEntityManagerFactory) {
        return new JpaTransactionManager(departmentEntityManagerFactory);
    }
}
```

4. Using Multiple Data Sources in Your Application

With the above configurations, Spring Boot knows which DataSource to use for each set of repositories and entities. The @Primary annotation is used to indicate the primary DataSource when only one is needed in a particular context.

Exercise 10: Employee Management System – Hibernate-Specific Features

To leverage Hibernate-specific features in your Employee Management System, you can use Hibernate-specific annotations to customize entity mappings, configure the Hibernate dialect and properties for optimal performance, and implement batch processing for bulk operations. Below are detailed instructions on how to accomplish these tasks.

1. Hibernate-Specific Annotations

Hibernate offers specific annotations that can help you customize your entity mappings beyond the standard JPA annotations.

1.1 @BatchSize Annotation

This annotation helps reduce the number of queries by fetching related entities in batches.

Example:

java

Copy code

```
import org.hibernate.annotations.BatchSize;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import java.util.List;

@Entity
public class Department {

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

```
    private String name;

    @OneToMany(mappedBy = "department")
    @BatchSize(size = 10) // Fetch 10 employees at a time
    private List<Employee> employees;

    // Getters and Setters
}
```

1.2 @Fetch Annotation

The @Fetch annotation allows you to control how Hibernate fetches associations. You can specify strategies like JOIN, SELECT, or SUBSELECT.

Example:

java

Copy code

```
import org.hibernate.annotations.Fetch;
import org.hibernate.annotations.FetchMode;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import java.util.List;

@Entity
public class Department {

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

    private String name;

    @OneToMany(mappedBy = "department")
    @Fetch(FetchMode.JOIN) // Fetch employees using a join query
    private List<Employee> employees;

    // Getters and Setters
}
```

1.3 @Type Annotation

Use the @Type annotation to map non-standard database types to Java types.

Example:

java

Copy code

```
import org.hibernate.annotations.Type;

import javax.persistence.Entity;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import java.util.UUID;

@Entity
public class Employee {

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

    private String name;

    @Type(type = "uuid-char")
    private UUID uniqueId;

    // Getters and Setters
}
```

2. Configuring Hibernate Dialect and Properties

The Hibernate dialect determines how Hibernate generates SQL statements for a particular database. Configuring the dialect and other Hibernate properties can significantly affect performance.

2.1 Configure Hibernate Dialect

In your application.properties file, set the Hibernate dialect according to the database you're using.

For MySQL:

properties

Copy code

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

For PostgreSQL:

properties

Copy code

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

2.2 Configure Other Hibernate Properties

You can configure various properties to optimize performance, such as caching, batch fetching, and query plans.

Example properties in application.properties:

properties

Copy code

```
# Enable second-level cache
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.jcache.JCacheRegion
Factory

# Configure batch fetching
spring.jpa.properties.hibernate.default_batch_fetch_size=16

# Enable SQL comments for debugging
spring.jpa.properties.hibernate.use_sql_comments=true
```

```
# Use JDBC batch processing
spring.jpa.properties.hibernate.jdbc.batch_size=20
spring.jpa.properties.hibernate.order_inserts=true
spring.jpa.properties.hibernate.order_updates=true
```

3. Batch Processing

Hibernate allows you to perform batch processing for bulk operations, which is useful when dealing with large volumes of data.

3.1 Configuring Batch Processing

Configure batch size in your application.properties:

properties

Copy code

```
spring.jpa.properties.hibernate.jdbc.batch_size=20
```

This setting determines how many SQL statements Hibernate will batch together for execution.

3.2 Implementing Batch Processing in Code

When performing batch operations (e.g., inserting or updating multiple entities), you should handle transactions and entity states properly to avoid issues like the OutOfMemoryError.

Example of batch insertion:

java

Copy code

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.persistence.EntityManager;
import java.util.List;

@Service
public class EmployeeService {

   @Autowired
   private EntityManager entityManager;

   @Transactional
   public void batchInsertEmployees(List<Employee> employees) {
      int batchSize = 20;
      for (int i = 0; i < employees.size(); i++) {
         entityManager.persist(employees.get(i));
         if (i % batchSize == 0 && i > 0) {
            entityManager.flush();
            entityManager.clear();
         }
      }
      entityManager.flush();
      entityManager.clear();
   }}
```

This example uses EntityManager to persist employees in batches, flushing and clearing the persistence context periodically to prevent memory overflow.