Exercise 1: Employee Management System - Overview and Setup

Step 1: Create a Spring Boot Project

- Use the Spring Initialiser to create a new Spring Boot project named `EmployeeManagementSystem`.
- Add the following dependencies: Spring Data JPA, H2 Database, Spring Web, Lombok.

```
spring init --name=EmployeeManagementSystem --dependencies=web,data-jpa,h2,lombok EmployeeManagementSystem
```

Step 2: Configure Application Properties

- Configure the `application.properties` file to set up the H2 database connection.

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Exercise 2: Employee Management System - Creating Entities

Step 1: Define the Employee Entity

- Create an `Employee` entity with fields `id`, `name`, `email`, and `department`.
- Use JPA annotations like `@Entity`, `@Table`, `@Id`, and `@GeneratedValue`.

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

import javax.persistence.*;

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

    @Id
```

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

    private String name;
    private String email;

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

    // Getters and Setters
}
```

Step 2: Define the Department Entity

- Create a `Department` entity with fields `id` and `name`.
- Define a one-to-many relationship between `Department` and `Employee`.

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

import javax.persistence.*;
import java.util.Set;

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

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

    private String name;

    @OneToMany(mappedBy = "department")
    private Set<Employee> employees;

    // Getters and Setters
}
```

Exercise 3: Employee Management System - Creating Repositories

Step 1: Create EmployeeRepository Interface

- Extend the `JpaRepository` interface to create a repository for the `Employee` entity.
- Define derived query methods using method name conventions.

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

Step 2: Create DepartmentRepository Interface

- Similarly, create a repository for the `Department` entity.

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

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

import java.util.List;

public interface DepartmentRepository extends
JpaRepository<Department, Long> {
    // Derived query method
    List<Department> findByName(String name);
}
```

Exercise 4: Employee Management System - Implementing CRUD Operations

Step 1: Implement CRUD Operations in EmployeeController

- Create a REST controller for `Employee` with endpoints for CRUD operations using `JpaRepository` methods.

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

import com.example.employeemanagementsystem.entity.Employee;
import
com.example.employeemanagementsystem.repository.EmployeeRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

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

    @Autowired
    private EmployeeRepository employeeRepository;

    @GetMapping
    public List<Employee> getAllEmployees() {
        return employeeRepository.findAll();
    }

    @PostMapping
    public Employee createEmployee(@RequestBody Employee employee) {
        return employeeRepository.save(employee);
    }

    @PutMapping("/{id}")
    public Employee updateEmployee(@PathVariable Long id,
@RequestBody Employee employeeDetails) {
        Employee employee =
employeeRepository.findById(id).orElseThrow();
        employee.setName(employeeDetails.getName());
        employee.setEmail(employeeDetails.getEmail());
        employee.setDepartment(employeeDetails.getDepartment());
```

```
        return employeeRepository.save(employee);
    }

    @DeleteMapping("/{id}")
    public void deleteEmployee(@PathVariable Long id) {
        employeeRepository.deleteById(id);
    }
}
```

Step 2: Implement CRUD Operations in DepartmentController

- Similarly, create a REST controller for Department with endpoints for CRUD operations.

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

import com.example.employeemanagementsystem.entity.Department;
import
com.example.employeemanagementsystem.repository.DepartmentRepository
;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

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

    @Autowired
    private DepartmentRepository departmentRepository;

    @GetMapping
    public List<Department> getAllDepartments() {
        return departmentRepository.findAll();
    }

    @PostMapping
    public Department createDepartment(@RequestBody Department
department) {
        return departmentRepository.save(department);
```

```
    }

    @PutMapping("/{id}")
    public Department updateDepartment(@PathVariable Long id,
@RequestBody Department departmentDetails) {
        Department department =
departmentRepository.findById(id).orElseThrow();
        department.setName(departmentDetails.getName());
        return departmentRepository.save(department);
    }

    @DeleteMapping("/{id}")
    public void deleteDepartment(@PathVariable Long id) {
        departmentRepository.deleteById(id);
    }
}
```

Exercise 5: Employee Management System - Defining Query Methods

Step 1: Define Custom Query Methods Using @Query Annotation

- Use the @Query annotation to define custom queries in the repository interface.

```
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 java.util.List;

public interface EmployeeRepository extends JpaRepository<Employee,
Long> {

    List<Employee> findByName(String name);

    @Query("SELECT e FROM Employee e WHERE e.email = ?1")
    Employee findByEmail(String email);
}
```

Step 2: Define Named Queries in the Employee Entity

- Use `@NamedQuery` to define named queries in the entity class.

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

import javax.persistence.*;

@Entity
@NamedQuery(name = "Employee.findByDepartmentName",
            query = "SELECT e FROM Employee e WHERE
e.department.name = :departmentName")
@Table(name = "employees")
public class Employee {
    // Fields, Getters, Setters, etc.
}
```

Exercise 6: Employee Management System - Implementing Pagination and Sorting

Step 1: Implement Pagination and Sorting in EmployeeController

- Modify the `EmployeeController` to include pagination and sorting in the `GET` request.

```
@GetMapping("/paged")
public Page<Employee> getAllEmployeesPaged(
        @RequestParam int page,
        @RequestParam int size,
        @RequestParam String sort) {
    Pageable pageable = PageRequest.of(page, size, Sort.by(sort));
    return employeeRepository.findAll(pageable);
}
```

Exercise 7: Employee Management System - Enabling Entity Auditing

Step 1: Enable Auditing in the Application

- Enable auditing in the main application class using `@EnableJpaAuditing`.

```
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);
    }
}
```

Step 2: Add Audit Fields to the Employee Entity

- Add auditing fields to the `Employee` entity using annotations like `@CreatedDate` and `@LastModifiedDate`.

```
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import java.time.LocalDateTime;

@Entity
public class Employee {

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

    private String name;
    private String email;
```

```
    @CreatedDate
    private LocalDateTime createdDate;

    @LastModifiedDate
    private LocalDateTime lastModifiedDate;

    // Other fields, Getters, Setters
}
```

Exercise 8: Employee Management System - Creating Projections

Step 1: Define Interface-Based Projections

- Define an interface-based projection to fetch specific fields from the `Employee` entity.

```
public interface EmployeeProjection {
    String getName();
    String getEmail();
}
```

Step 2: Use the Projection in Repository

- Use the projection in the repository interface to retrieve selected fields.

```
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 java.util.List;

public interface EmployeeRepository extends JpaRepository<Employee,
Long> {

    @Query("SELECT e.name AS name, e.email AS email FROM Employee
e")
    List<EmployeeProjection> findAllEmployeeNamesAndEmails();
```

```
}
```

Exercise 9: Employee Management System - Customizing Data Source Configuration

Step 1: Configure Multiple Data Sources

- Configure multiple data sources in the `application.properties` file.

```
spring.datasource.primary.url=jdbc:h2:mem:primarydb
spring.datasource.primary.username=sa
spring.datasource.primary.password=password
spring.datasource.primary.driverClassName=org.h2.Driver

spring.datasource.secondary.url=jdbc:h2:mem:secondarydb
spring.datasource.secondary.username=sa
spring.datasource.secondary.password=password
spring.datasource.secondary.driverClassName=org.h2.Driver
```

Exercise 10: Employee Management System - Hibernate-Specific Features

Step 1: Use Hibernate-Specific Annotations

- Utilize Hibernate-specific annotations like `@BatchSize` and `@Cache` in the `Employee` entity.

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

import org.hibernate.annotations.BatchSize;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

import javax.persistence.*;

@Entity
@Table(name = "employees")
@BatchSize(size = 10)
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Employee {
```

```
    // Fields, Getters, Setters
}
```

Step 2: Configure Hibernate Properties

- Configure Hibernate-specific properties in the `application.properties` file.

```
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.use_sql_comments=true
spring.jpa.properties.hibernate.generate_statistics=true
```

Step 3: Implement Batch Processing

- Implement batch processing in the service layer to optimize large-scale updates.

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

import com.example.employeemanagementsystem.entity.Employee;
import org.springframework.stereotype.Service;

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

@Service
public class EmployeeService {

    private final EntityManager entityManager;

    public EmployeeService(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    @Transactional
    public void batchUpdateEmployees(List<Employee> employees) {
        for (int i = 0; i < employees.size(); i++) {
            entityManager.merge(employees.get(i));
            if (i % 20 == 0) { // Flush every 20 updates
                entityManager.flush();
                entityManager.clear();
```

```
                }
            }
        }
    }
```