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

**Business Scenario:**

You are developing an employee management system that will manage employee data, departments, and their relationships.

1. **Creating a Spring Boot Project:**
   * Initialize a new Spring Boot project named **EmployeeManagementSystem**.
   * Add dependencies: **Spring Data JPA, H2 Database, Spring Web, Lombok**.
2. **Configuring Application Properties:**
   * Configure **application.properties** for 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**

**Business Scenario:**

Define JPA entities for Employee and Department with appropriate relationships.

1. **Creating JPA Entities:**
   * Define **Employee** entity with fields: **id, name, email, department**.
   * Define **Department** entity with fields: **id, name**.
2. **Mapping Entities to Database Tables:**
   * Use annotations like **@Entity, @Table, @Id, @GeneratedValue**, etc.
   * Define one-to-many relationship between **Department** and **Employee**.

**//Employee.entity:**

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.\*;

@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")

private Department department;

}

//**Department.entity**

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.\*;

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;

}

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

**Business Scenario:**

Create repositories for Employee and Department entities to perform CRUD operations.

1. **Overview of Spring Data Repositories:**
   * Learn the benefits of using Spring Data repositories.
2. **Creating Repositories:**
   * Create **EmployeeRepository** and **DepartmentRepository** interfaces extending **JpaRepository**.
   * Define derived query methods in these repositories.

**//EmployeeRepository**

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<Employee> findByName(String name);

}

**//DepartmentRepository**

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

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface DepartmentRepository extends JpaRepository<Department, Long> {

List<Department> findByName(String name);

}

**Exercise 4: Employee Management System - Implementing CRUD Operations**

**Business Scenario:**

Implement CRUD operations for managing employees and departments.

1. **Basic CRUD Operations:**
   * Use **JpaRepository** methods to create, read, update, and delete employees and departments.
   * Implement RESTful endpoints for these operations using **EmployeeController** and **DepartmentController**.

**//EmployeeController**

import lombok.RequiredArgsConstructor;

import org.springframework.http.ResponseEntity;

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

@RestController

@RequestMapping("/employees")

@RequiredArgsConstructor

public class EmployeeController {

private final EmployeeRepository employeeRepository;

@PostMapping

public Employee createEmployee(@RequestBody Employee employee) {

return employeeRepository.save(employee);

}

@GetMapping("/{id}")

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

return employeeRepository.findById(id)

.map(ResponseEntity::ok)

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

}

@PutMapping("/{id}")

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

return employeeRepository.findById(id)

.map(existingEmployee -> {

employee.setId(id);

employeeRepository.save(employee);

return ResponseEntity.ok(employee);

})

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

}

@DeleteMapping("/{id}")

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

return employeeRepository.findById(id)

.map(employee -> {

employeeRepository.delete(employee);

return ResponseEntity.noContent().build();

})

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

}

}

**//DepartmentController**

import lombok.RequiredArgsConstructor;

import org.springframework.http.ResponseEntity;

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

@RestController

@RequestMapping("/departments")

@RequiredArgsConstructor

public class DepartmentController {

private final DepartmentRepository departmentRepository;

@PostMapping

public Department createDepartment(@RequestBody Department department) {

return departmentRepository.save(department);

}

@GetMapping("/{id}")

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

return departmentRepository.findById(id)

.map(ResponseEntity::ok)

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

}

@PutMapping("/{id}")

public ResponseEntity<Department> updateDepartment(@PathVariable Long id, @RequestBody Department department) {

return departmentRepository.findById(id)

.map(existingDepartment -> {

department.setId(id);

departmentRepository.save(department);

return ResponseEntity.ok(department);

})

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

}

@DeleteMapping("/{id}")

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

return departmentRepository.findById(id)

.map(department -> {

departmentRepository.delete(department);

return ResponseEntity.noContent().build();

})

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

}

}

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

**Business Scenario:**

Enhance your repository to support custom queries.

1. **Defining Query Methods:**
   * Use keywords in method names to create custom query methods.
   * Implement custom query methods using the **@Query** annotation.
2. **Named Queries:**
   * Define and execute named queries with **@NamedQuery** and **@NamedQueries**.

**//Custom Query Methods in EmployeeRepository:**

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

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

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

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByName(String name);

@Query("SELECT e FROM Employee e WHERE e.email = :email")

Employee findByEmail(@Param("email") String email);

}

Named Queries in Employee Entity:

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.\*;

@Entity

@Table(name = "employees")

@NamedQuery(name = "Employee.findByEmail", query = "SELECT e FROM Employee e WHERE e.email = :email")

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

}

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

**Business Scenario:**

Add pagination and sorting capabilities to your employee search functionality.

1. **Pagination:**
   * Implement pagination for the employee list using **Page** and **Pageable**.
2. **Sorting:**
   * Add sorting functionality to your queries.
   * Combine pagination and sorting in your search endpoint.

**Modify EmployeeController for Pagination and Sorting:**

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

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

@RestController

@RequestMapping("/employees")

@RequiredArgsConstructor

public class EmployeeController {

private final EmployeeRepository employeeRepository;

@GetMapping

public Page<Employee> getAllEmployees(Pageable pageable) {

return employeeRepository.findAll(pageable);

}

}

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

**Business Scenario:**

Implement auditing to track the creation and modification of employees and departments.

1. **Entity Auditing:**
   * Enable auditing in your application by configuring auditing properties.
   * Use annotations like **@CreatedBy, @LastModifiedBy, @CreatedDate**, and **@LastModifiedDate**.

**Main Application Class:**

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);

}

}

**Modify Employee Entity:**

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import org.springframework.data.annotation.CreatedBy;

import org.springframework.data.annotation.CreatedDate;

import org.springframework.data.annotation.LastModifiedBy;

import org.springframework.data.annotation.LastModifiedDate;

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

import javax.persistence.\*;

import java.time.LocalDateTime;

@Entity

@EntityListeners(AuditingEntityListener.class)

@Table(name = "employees")

@Data

@NoArgsConstructor

@AllArgsConstructor

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;

@CreatedBy

private String createdBy;

@LastModifiedBy

private String lastModifiedBy;

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

}

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

**Business Scenario:**

Create projections to fetch specific data subsets from the employee and department entities.

1. **Projections:**
   * Define interface-based and class-based projections.
   * Use **@Value** and constructor expressions to control the fetched data.

**Define Interface-Based Projection:**

public interface EmployeeProjection {

String getName();

String getEmail();

}

Use Projection in Repository:

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

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

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

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Method to find employees by name

List<Employee> findByName(String name);

// Custom query method to find employee by email using named query

@Query("SELECT e FROM Employee e WHERE e.email = :email")

Employee findByEmail(@Param("email") String email);

// Method to fetch specific data using projection

@Query("SELECT e FROM Employee e")

List<EmployeeProjection> findAllEmployeeProjections();

}

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

**Business Scenario:**

Customize your data source configuration and manage multiple data sources.

Instructions:

1. **Spring Boot Auto-Configuration**:
   * Leverage Spring Boot auto-configuration for data sources.
2. **Externalizing Configuration:**
   * Externalize configuration with application.properties.
   * Manage multiple data sources within your application.

** Spring Boot Auto-Configuration:**

* Spring Boot auto-configures a single data source by default. To manage multiple data sources, you need to configure them manually.

** Externalizing Configuration:**

* application.properties Configuration for Multiple Data Sources:

**# Primary Data Source**

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

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

spring.datasource.primary.username=sa

spring.datasource.primary.password=password

**# Secondary Data Source**

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

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

spring.datasource.secondary.username=sa

spring.datasource.secondary.password=password

**Configure Multiple Data Sources in Java Configuration:**

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

import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

import org.springframework.jdbc.datasource.DataSourceTransactionManager;

import org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy;

import org.springframework.orm.jpa.JpaTransactionManager;

import org.springframework.orm.jpa.JpaVendorAdapter;

import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

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

import org.springframework.transaction.PlatformTransactionManager;

import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

import java.util.HashMap;

import java.util.Map;

@Configuration

@EnableTransactionManagement

public class DataSourceConfig {

@Bean

@Primary

@ConfigurationProperties("spring.datasource.primary")

public DataSourceProperties primaryDataSourceProperties() {

return new DataSourceProperties();

}

@Bean

@Primary

public DataSource primaryDataSource() {

return primaryDataSourceProperties().initializeDataSourceBuilder().build();

}

@Bean

@ConfigurationProperties("spring.datasource.secondary")

public DataSourceProperties secondaryDataSourceProperties() {

return new DataSourceProperties();

}

@Bean

public DataSource secondaryDataSource() {

return secondaryDataSourceProperties().initializeDataSourceBuilder().build();

}

@Bean

@Primary

public LocalContainerEntityManagerFactoryBean entityManagerFactory(@Qualifier("primaryDataSource") DataSource dataSource) {

LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();

em.setDataSource(dataSource);

em.setPackagesToScan("com.example.domain");

JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

em.setJpaVendorAdapter(vendorAdapter);

Map<String, Object> properties = new HashMap<>();

properties.put("hibernate.hbm2ddl.auto", "update");

properties.put("hibernate.dialect", "org.hibernate.dialect.H2Dialect");

em.setJpaPropertyMap(properties);

return em;

}

@Bean

@Primary

public PlatformTransactionManager transactionManager(@Qualifier("primaryDataSource") DataSource dataSource) {

JpaTransactionManager transactionManager = new JpaTransactionManager();

transactionManager.setEntityManagerFactory(entityManagerFactory(dataSource).getObject());

return transactionManager;

}

}

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

**Business Scenario:**

Leverage Hibernate-specific features to enhance your application's performance and capabilities.

1. **Hibernate-Specific Annotations:**
   * Use Hibernate-specific annotations to customize entity mappings.
2. **Configuring Hibernate Dialect and Properties:**
   * Configure Hibernate dialect and properties for optimal performance.
3. **Batch Processing:**
   * Implement batch processing with Hibernate for bulk operations.

**Use Hibernate-Specific Annotations:**

import org.hibernate.annotations.BatchSize;

import org.hibernate.annotations.Fetch;

import org.hibernate.annotations.FetchMode;

import org.hibernate.annotations.LazyCollection;

import org.hibernate.annotations.LazyCollectionType;

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "department\_id")

@Fetch(FetchMode.JOIN)

private Department department;

// Other fields, getters, and setters

}

application.properties Configuration:

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

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

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

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

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

**Implement Batch Processing in a Service Class:**

import lombok.RequiredArgsConstructor;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import javax.persistence.Query;

import java.util.List;

@Service

@RequiredArgsConstructor

public class EmployeeService {

@PersistenceContext

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();

}

}

}

}