**PAKHI SHARMA**

**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.

**Instructions:**

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*

**pom.xml dependencies**

xml

CopyEdit

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

**application.properties**

properties

CopyEdit

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

spring.h2.console.enabled=true

spring.jpa.show-sql=true

**Exercise 2: Employee Management System - Creating Entities**

**Business Scenario:**

Define JPA entities for Employee and Department with appropriate relationships.

**Instructions:**

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**.

@Entity

@Data

@NoArgsConstructor

@AllArgsConstructor

@Table(name = "departments")

public class Department {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

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

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

}

@Entity

@Data

@NoArgsConstructor

@AllArgsConstructor

@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;

}

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

**Business Scenario:**

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

**Instructions:**

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.

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByDepartmentName(String name);

}

public interface DepartmentRepository extends JpaRepository<Department, Long> {

Optional<Department> findByName(String name);

}

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

**Business Scenario:**

Implement CRUD operations for managing employees and departments.

**Instructions:**

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**.

@RestController

@RequestMapping("/employees")

@RequiredArgsConstructor

public class EmployeeController {

private final EmployeeRepository employeeRepo;

private final DepartmentRepository deptRepo;

@PostMapping

public Employee create(@RequestBody Employee emp) {

return employeeRepo.save(emp);

}

@GetMapping

public List<Employee> list() {

return employeeRepo.findAll();

}

@PutMapping("/{id}")

public Employee update(@PathVariable Long id, @RequestBody Employee emp) {

emp.setId(id);

return employeeRepo.save(emp);

}

@DeleteMapping("/{id}")

public void delete(@PathVariable Long id) {

employeeRepo.deleteById(id);

}

}

@RestController

@RequestMapping("/departments")

@RequiredArgsConstructor

public class DepartmentController {

private final DepartmentRepository deptRepo;

@PostMapping

public Department create(@RequestBody Department dept) {

return deptRepo.save(dept);

}

}

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

**Business Scenario:**

Enhance your repository to support custom queries.

**Instructions:**

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**.

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByNameContaining(String keyword);

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

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

}

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

**Business Scenario:**

Add pagination and sorting capabilities to your employee search functionality.

**Instructions:**

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.

@GetMapping("/search")

public Page<Employee> search(@RequestParam int page, @RequestParam int size,

@RequestParam String sortBy) {

return employeeRepo.findAll(PageRequest.of(page, size, Sort.by(sortBy)));

}

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

**Business Scenario:**

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

**Instructions:**

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

@SpringBootApplication

@EnableJpaAuditing

public class EmployeeManagementSystemApplication { }

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

**Business Scenario:**

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

**Instructions:**

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

@MappedSuperclass

@EntityListeners(AuditingEntityListener.class)

@Data

public abstract class Auditable {

@CreatedDate

@Column(updatable = false)

private LocalDateTime createdDate;

@LastModifiedDate

private LocalDateTime modifiedDate;

}

**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.

public interface EmployeeNameEmailProjection {

String getName();

String getEmail();

}

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<EmployeeNameEmailProjection> findAllBy();

}

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

**Business Scenario:**

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

**Instructions:**

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.

@Entity

@BatchSize(size = 20) // Hibernate batch fetching

public class Employee { ... }