**Week 3 assignment for cognizant**

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

#### **1. Creating a Spring Boot Project:**

* **Initialize a Spring Boot Project**:
  + Use a tool like Spring Initializr (<https://start.spring.io/>) to create a new project.
  + Set the project name as EmployeeManagementSystem.
  + Add the following dependencies:
    - **Spring Data JPA**: To interact with the database.
    - **H2 Database**: An in-memory database for development and testing.
    - **Spring Web**: For building web applications and RESTful services.
    - **Lombok**: To reduce boilerplate code like getters, setters, etc.

#### **2. Configuring Application Properties:**

* **application.properties**:
  + Configure the database connection in the src/main/resources/application.properties file:

properties

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 # Optional: Enables H2 console for manual inspection

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

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

* **Employee Entity**:
  + Fields: id, name, email, department.
  + Example:

@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")  
 private Department department;  
  
 // Getters and Setters  
}

* **Department Entity**:
  + Fields: id, name.
  + Example:

@Entity  
@Table(name = "departments")  
public class Department {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
  
 private String name;  
  
 @OneToMany(mappedBy = "department", cascade = CascadeType.ALL, orphanRemoval = true)  
 private List<Employee> employees = new ArrayList<>();  
  
 // Getters and Setters  
}

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

#### **1. Overview of Spring Data Repositories:**

* **Benefits**:
  + Simplifies database interactions.
  + Provides built-in CRUD operations.
  + Supports custom query methods.

#### **2. Creating Repositories:**

* **EmployeeRepository**:

public interface EmployeeRepository extends JpaRepository<Employee, Long> {  
 // Define custom query methods if needed  
}

* **DepartmentRepository**:

public interface DepartmentRepository extends JpaRepository<Department, Long> {  
 // Define custom query methods if needed  
}

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

#### **1. Basic CRUD Operations:**

* **EmployeeController**:

@RestController  
@RequestMapping("/api/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 ResponseEntity<Employee> updateEmployee(@PathVariable Long id, @RequestBody Employee employeeDetails) {  
 Employee employee = employeeRepository.findById(id)  
 .orElseThrow(() -> new ResourceNotFoundException("Employee not found"));  
 employee.setName(employeeDetails.getName());  
 employee.setEmail(employeeDetails.getEmail());  
 employee.setDepartment(employeeDetails.getDepartment());  
 Employee updatedEmployee = employeeRepository.save(employee);  
 return ResponseEntity.ok(updatedEmployee);  
 }  
  
 @DeleteMapping("/{id}")  
 public ResponseEntity<Map<String, Boolean>> deleteEmployee(@PathVariable Long id) {  
 Employee employee = employeeRepository.findById(id)  
 .orElseThrow(() -> new ResourceNotFoundException("Employee not found"));  
 employeeRepository.delete(employee);  
 Map<String, Boolean> response = new HashMap<>();  
 response.put("deleted", Boolean.TRUE);  
 return ResponseEntity.ok(response);  
 }  
}

* **DepartmentController**:

@RestController  
@RequestMapping("/api/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 ResponseEntity<Department> updateDepartment(@PathVariable Long id, @RequestBody Department departmentDetails) {  
 Department department = departmentRepository.findById(id)  
 .orElseThrow(() -> new ResourceNotFoundException("Department not found"));  
 department.setName(departmentDetails.getName());  
 Department updatedDepartment = departmentRepository.save(department);  
 return ResponseEntity.ok(updatedDepartment);  
 }  
  
 @DeleteMapping("/{id}")  
 public ResponseEntity<Map<String, Boolean>> deleteDepartment(@PathVariable Long id) {  
 Department department = departmentRepository.findById(id)  
 .orElseThrow(() -> new ResourceNotFoundException("Department not found"));  
 departmentRepository.delete(department);  
 Map<String, Boolean> response = new HashMap<>();  
 response.put("deleted", Boolean.TRUE);  
 return ResponseEntity.ok(response);  
 }  
}

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

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

* **Custom Query Methods**:
  + You can create custom query methods by simply defining method signatures in your repository interfaces. Spring Data JPA will automatically generate the queries based on the method names.
  + **Example** in EmployeeRepository:

public interface EmployeeRepository extends JpaRepository<Employee, Long> {  
 List<Employee> findByDepartmentName(String departmentName);  
 List<Employee> findByNameContaining(String name);  
}

* + **Explanation**:
    - findByDepartmentName: Fetches all employees belonging to a department with a specific name.
    - findByNameContaining: Fetches all employees whose names contain a specific substring.

#### **2. Using @Query Annotation:**

* **Custom JPQL Queries**:
  + Use the @Query annotation to write custom JPQL or native SQL queries.
  + **Example** in EmployeeRepository:

@Query("SELECT e FROM Employee e WHERE e.department.name = ?1")  
List<Employee> findEmployeesByDepartmentName(String departmentName);  
  
@Query("SELECT e FROM Employee e WHERE e.name LIKE %?1%")  
List<Employee> searchByName(String name);

#### **3. Named Queries:**

* **Using Named Queries**:
  + You can define named queries using the @NamedQuery annotation at the entity level.
  + **Example** in Employee Entity:

@Entity  
@NamedQuery(name = "Employee.findByDepartmentName", query = "SELECT e FROM Employee e WHERE e.department.name = :name")  
public class Employee {  
 // Entity fields and methods  
}

* + **Usage** in EmployeeRepository:

List<Employee> findByDepartmentName(@Param("name") String departmentName);

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

#### **1. Pagination:**

* **Implementing Pagination**:
  + Spring Data JPA provides the Page and Pageable interfaces for pagination.
  + **Example** in EmployeeRepository:

Page<Employee> findAll(Pageable pageable);

* + **Usage** in EmployeeController:

@GetMapping("/paginated")  
public Page<Employee> getAllEmployeesPaginated(@RequestParam int page, @RequestParam int size) {  
 Pageable pageable = PageRequest.of(page, size);  
 return employeeRepository.findAll(pageable);  
}

#### **2. Sorting:**

* **Adding Sorting**:
  + Sorting can be applied along with pagination or separately.
  + **Example** in EmployeeRepository:

List<Employee> findAllByOrderByNameAsc();

* + **Combined Pagination and Sorting** in EmployeeController:

@GetMapping("/sorted-paginated")  
public Page<Employee> getAllEmployeesSortedAndPaginated(@RequestParam int page, @RequestParam int size, @RequestParam String sortBy) {  
 Pageable pageable = PageRequest.of(page, size, Sort.by(sortBy).ascending());  
 return employeeRepository.findAll(pageable);  
}

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

#### **1. Entity Auditing:**

* **Enable Auditing**:
  + Enable JPA Auditing in your Spring Boot application by adding the @EnableJpaAuditing annotation to your main application class.
  + **Example** in EmployeeManagementSystemApplication:

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

* **Audit Fields**:
  + Add auditing fields to your entities using @CreatedDate, @LastModifiedDate, @CreatedBy, and @LastModifiedBy.
  + **Example** in Employee Entity:

@Entity  
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")  
 private Department department;  
  
 @CreatedDate  
 @Column(updatable = false)  
 private LocalDateTime createdDate;  
  
 @LastModifiedDate  
 private LocalDateTime lastModifiedDate;  
  
 // Getters and Setters  
}

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

#### **1. Projections:**

* **Interface-based Projections**:
  + Define an interface that only includes the fields you want to fetch.
  + **Example**:

public interface EmployeeNameOnly {  
 String getName();  
}

* + **Usage** in EmployeeRepository:

List<EmployeeNameOnly> findBy();

* **Class-based Projections**:
  + Use DTO (Data Transfer Object) to fetch specific data.
  + **Example**:

public class EmployeeDTO {  
 private String name;  
 private String departmentName;  
  
 public EmployeeDTO(String name, String departmentName) {  
 this.name = name;  
 this.departmentName = departmentName;  
 }  
  
 // Getters and Setters  
}

* + **Usage** in EmployeeRepository:

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

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

#### **1. Spring Boot Auto-Configuration:**

* **Customizing Data Source**:
  + You can override the default data source configuration by defining beans for DataSource, EntityManagerFactory, etc.
  + **Example**:

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

#### **2. Externalizing Configuration:**

* **Using application.properties**:
  + Externalize configuration like data source URL, username, and password in application.properties.
  + **Example**:

properties

spring.datasource.url=jdbc:mysql://localhost:3306/employees\_db  
spring.datasource.username=root  
spring.datasource.password=secret

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

* **Multiple Data Sources**:
  + Define multiple DataSource beans and specify which one is primary.
  + **Example**:

@Bean  
@Primary  
@ConfigurationProperties(prefix = "spring.datasource.primary")  
public DataSource primaryDataSource() {  
 return DataSourceBuilder.create().build();  
}  
  
@Bean  
@ConfigurationProperties(prefix = "spring.datasource.secondary")  
public DataSource secondaryDataSource() {  
 return DataSourceBuilder.create().build();  
}

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

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

* **Using Hibernate Annotations**:
  + Use Hibernate-specific annotations for advanced mapping configurations, such as @NaturalId, @BatchSize, etc.
  + **Example** in Employee Entity:

@Entity  
public class Employee {  
  
 @NaturalId  
 private String email;  
  
 @BatchSize(size = 10)  
 @OneToMany(mappedBy = "employee")  
 private Set<Address> addresses;  
  
 // Other fields  
}

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

* **Hibernate Dialect**:
  + Specify the Hibernate dialect in your application.properties for optimized SQL generation.
  + **Example**:

properties

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

#### **3. Batch Processing:**

* **Implementing Batch Processing**:
  + Optimize bulk operations by using batch processing features.
  + **Example**:

properties

spring.jpa.properties.hibernate.jdbc.batch\_size=20  
spring.jpa.properties.hibernate.order\_inserts=true  
spring.jpa.properties.hibernate.order\_updates=true

* + **In your service layer**:

public void saveAllEmployees(List<Employee> employees) {  
 employeeRepository.saveAll(employees);  
}