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

// **EmployeeManagementSystemApplication**.java

package com.example.EmployeeManagementSystem;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class EmployeeManagementSystemApplication {

public static void main(String[] args) {

SpringApplication.run(EmployeeManagementSystemApplication.class, args);

}

}

**// Employee.java (in model package)**

package com.example.EmployeeManagementSystem.model;

import lombok.Data;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

@Data

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String department;

}

**// EmployeeRepository.java (in repository package)**

package com.example.EmployeeManagementSystem.repository;

import com.example.EmployeeManagementSystem.model.Employee;

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

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}

**// EmployeeController.java (in controller package)**

package com.example.EmployeeManagementSystem.controller;

import com.example.EmployeeManagementSystem.model.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 addEmployee(@RequestBody Employee employee) {

return employeeRepository.save(employee);

}

@GetMapping("/{id}")

public Employee getEmployeeById(@PathVariable Long id) {

return employeeRepository.findById(id).orElse(null);

}

@DeleteMapping("/{id}")

public void deleteEmployee(@PathVariable Long id) {

employeeRepository.deleteById(id);

}

}

**# application.properties (in src/main/resources)**

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

<dependencies>

<!-- Spring Boot Starter Web -->

<dependency>

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

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

</dependency>

<!-- Spring Boot Starter Data JPA -->

<dependency>

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

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

</dependency>

<!-- H2 Database -->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<!-- Lombok -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.24</version>

<scope>provided</scope>

</dependency>

<!-- Spring Boot Starter Test -->

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

**Components of the Employee Management System**The Employee Management System is a Spring Boot application designed to manage employee data. It includes an Employee Management System Application class to start the application, an Employee entity representing employee data with fields like id, name, and department, and an Employee Repository interface for CRUD operations. The Employee Controller class provides RESTful endpoints to list, add, retrieve, and delete employees. Configuration is handled in application.properties to use an in-memory H2 database for simplicity. The pom.xml file manages dependencies like Spring Data JPA, H2 Database, and Lombok for boilerplate code reduction. This setup allows for efficient development and management of employee records through a simple REST AP

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

**// Department.java**

package com.example.EmployeeManagementSystem.model;

import lombok.Data;

import javax.persistence.\*;

import java.util.Set;

@Entity

@Table(name = "departments")

@Data

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

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

private Set<Employee> employees;

}

**// Employee.java**

package com.example.EmployeeManagementSystem.model;

import lombok.Data;

import javax.persistence.\*;

@Entity

@Table(name = "employees")

@Data

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;

}

### ****Explanation****

The Department entity represents departments and includes fields for id and name. It uses the @OneToMany annotation to define a one-to-many relationship with the Employee entity, indicating that one department can have multiple employees.

The Employee entity represents employees with fields for id, name, and email. It uses the @ManyToOne annotation to specify that each employee belongs to one department, linking to the Department entity via department\_id.

This setup maps both entities to their respective database tables and establishes a relationship where each department can have multiple employees.

**Exercise 3: Employee Management System - Creating Repositories**package com.example.yourapp.model;

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 firstName;

private String lastName;

private String email;

// Getters and setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

**Department Entity**package com.example.yourapp.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String location;

// Getters and setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getLocation() {

return location;

}

public void setLocation(String location) {

this.location = location;

}

}

**Repository Interfaces**

package com.example.yourapp.repository;

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

import org.springframework.stereotype.Repository;

import com.example.yourapp.model.Employee;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Derived query methods

Employee findByEmail(String email);

List<Employee> findByLastName(String lastName);

}

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

**DepartmentRepository**

package com.example.yourapp.repository;

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

import org.springframework.stereotype.Repository;

import com.example.yourapp.model.Department;

import java.util.List;

@Repository

public interface DepartmentRepository extends JpaRepository<Department, Long> {

// Derived query methods

Department findByName(String name);

List<Department> findByLocation(String location);

}

**Application Class**

package com.example.yourapp;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class YourAppApplication {

public static void main(String[] args) {

SpringApplication.run(YourAppApplication.class, args);

}

}

package com.example.yourapp.controller;

import com.example.yourapp.model.Employee;

import com.example.yourapp.repository.EmployeeRepository;

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

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeRepository employeeRepository;

@PostMapping

public ResponseEntity<Employee> createEmployee(@RequestBody Employee employee) {

Employee savedEmployee = employeeRepository.save(employee);

return new ResponseEntity<>(savedEmployee, HttpStatus.CREATED);

}

@GetMapping("/{id}")

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

Optional<Employee> employee = employeeRepository.findById(id);

return employee.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());

}

@GetMapping

public List<Employee> getAllEmployees() {

return employeeRepository.findAll();

}

@PutMapping("/{id}")

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

if (!employeeRepository.existsById(id)) {

return ResponseEntity.notFound().build();

}

employee.setId(id);

Employee updatedEmployee = employeeRepository.save(employee);

return ResponseEntity.ok(updatedEmployee);

}

@DeleteMapping("/{id}")

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

if (!employeeRepository.existsById(id)) {

return ResponseEntity.notFound().build();

}

employeeRepository.deleteById(id);

return ResponseEntity.noContent().build();

}

}

**DepartmentController**

package com.example.yourapp.controller;

import com.example.yourapp.model.Department;

import com.example.yourapp.repository.DepartmentRepository;

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

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/departments")

public class DepartmentController {

@Autowired

private DepartmentRepository departmentRepository;

@PostMapping

public ResponseEntity<Department> createDepartment(@RequestBody Department department) {

Department savedDepartment = departmentRepository.save(department);

return new ResponseEntity<>(savedDepartment, HttpStatus.CREATED);

}

@GetMapping("/{id}")

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

Optional<Department> department = departmentRepository.findById(id);

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

}

@GetMapping

public List<Department> getAllDepartments() {

return departmentRepository.findAll();

}

@PutMapping("/{id}")

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

if (!departmentRepository.existsById(id)) {

return ResponseEntity.notFound().build();

}

department.setId(id);

Department updatedDepartment = departmentRepository.save(department);

return ResponseEntity.ok(updatedDepartment);

}

@DeleteMapping("/{id}")

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

if (!departmentRepository.existsById(id)) {

return ResponseEntity.notFound().build();

}

departmentRepository.deleteById(id);

return ResponseEntity.noContent().build();

}}

In a Spring Boot application, CRUD operations for managing Employee and Department entities are implemented using JpaRepository methods. The EmployeeController and DepartmentController classes expose these operations through RESTful endpoints. These controllers handle HTTP requests to create, read, update, and delete employees and departments, providing a simple and standardized way to interact with the data via HTTP methods like POST, GET, PUT, and DELETE. This approach streamlines data management and integrates seamlessly with web clients.

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

// Employee Entity

package com.example.yourapp.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.NamedQuery;

@Entity

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

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String firstName;

private String lastName;

private String email;

// Getters and setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

// Department Entity

package com.example.yourapp.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.NamedQuery;

@Entity

@NamedQuery(name = "Department.findByName", query = "SELECT d FROM Department d WHERE d.name = :name")

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String location;

// Getters and setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getLocation() {

return location;

}

public void setLocation(String location) {

this.location = location;

}

}

// EmployeeRepository with Custom and Named Queries

package com.example.yourapp.repository;

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

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

import com.example.yourapp.model.Employee;

import java.util.List;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

Employee findByEmail(String email);

List<Employee> findByLastName(String lastName);

List<Employee> findByFirstNameAndLastName(String firstName, String lastName);

@Query("SELECT e FROM Employee e WHERE e.email = ?1")

Employee findEmployeeByEmail(String email);

@Query("SELECT e FROM Employee e WHERE e.lastName = ?1")

List<Employee> findEmployeesByLastName(String lastName);

@Query("SELECT e FROM Employee e WHERE e.firstName = ?1 AND e.lastName = ?2")

List<Employee> findEmployeesByFullName(String firstName, String lastName);

}

// DepartmentRepository with Custom and Named Queries

package com.example.yourapp.repository;

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

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

import com.example.yourapp.model.Department;

import java.util.List;

public interface DepartmentRepository extends JpaRepository<Department, Long> {

Department findByName(String name);

List<Department> findByLocation(String location);

@Query("SELECT d FROM Department d WHERE d.name = ?1")

Department findDepartmentByName(String name);

@Query("SELECT d FROM Department d WHERE d.location = ?1")

List<Department> findDepartmentsByLocation(String location);

}

In this setup, custom queries are defined for managing Employee and Department entities in a Spring Boot application. For simple queries, methods in the repository interfaces use keywords in their names to automatically generate SQL queries. For more complex queries, the @Query annotation allows you to write JPQL directly. Named queries are defined in the entity classes using @NamedQuery, enabling reusable, predefined queries that can be invoked via the repository methods. This approach offers flexibility and efficiency in querying the database.

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

package com.example.yourapp.repository;

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

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import com.example.yourapp.model.Employee;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

Page<Employee> findAll(Pageable pageable);

}

package com.example.yourapp.controller;

import com.example.yourapp.model.Employee;

import com.example.yourapp.repository.EmployeeRepository;

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.http.ResponseEntity;

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

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeRepository employeeRepository;

// Endpoint for paginated and sorted employee search

@GetMapping("/search")

public ResponseEntity<Page<Employee>> searchEmployees(

@RequestParam(defaultValue = "0") int page,

@RequestParam(defaultValue = "10") int size,

@RequestParam(defaultValue = "id") String sortBy,

@RequestParam(defaultValue = "asc") String sortDir) {

Pageable pageable = PageRequest.of(page, size, Sort.by(Sort.Direction.fromString(sortDir), sortBy));

Page<Employee> employees = employeeRepository.findAll(pageable);

return ResponseEntity.ok(employees);

}

}

To enhance employee search functionality with pagination and sorting, you can use Spring Data JPA's Page and Pageable interfaces. Pagination is implemented using findAll(Pageable pageable) in the repository to manage large datasets in manageable chunks. Sorting is incorporated by configuring the Pageable object with sorting parameters. The combined pagination and sorting functionality is then exposed through a REST endpoint, allowing clients to request specific pages and sort results according to their needs.

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

// JpaConfig.java

package com.example.yourapp.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.data.jpa.repository.config.EnableJpaAuditing;

@Configuration

@EnableJpaAuditing

public class JpaConfig {

// This class enables JPA Auditing

}

// Auditable.java

package com.example.yourapp.model;

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.Instant;

@MappedSuperclass

@EntityListeners(AuditingEntityListener.class)

public abstract class Auditable {

@CreatedDate

private Instant createdDate;

@LastModifiedDate

private Instant lastModifiedDate;

// Getters and setters

public Instant getCreatedDate() {

return createdDate;

}

public void setCreatedDate(Instant createdDate) {

this.createdDate = createdDate;

}

public Instant getLastModifiedDate() {

return lastModifiedDate;

}

public void setLastModifiedDate(Instant lastModifiedDate) {

this.lastModifiedDate = lastModifiedDate;

}

}

// Employee.java

package com.example.yourapp.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Employee extends Auditable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String firstName;

private String lastName;

private String email;

// Getters and setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

// Department.java

package com.example.yourapp.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Department extends Auditable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String location;

// Getters and setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getLocation() {

return location;

}

public void setLocation(String location) {

this.location = location;

}

}

// AuditorAwareImpl.java

package com.example.yourapp.config;

import org.springframework.data.domain.AuditorAware;

import org.springframework.stereotype.Component;

import java.util.Optional;

@Component

public class AuditorAwareImpl implements AuditorAware<String> {

@Override

public Optional<String> getCurrentAuditor() {

// Provide logic to return the current user's username

// For demonstration, using a static username

return Optional.of("system");

}

}

The setup for enabling entity auditing in a Spring Boot application involves several key components. The JpaConfig class activates JPA auditing with @EnableJpaAuditing, allowing the framework to track and manage audit information. The Auditable base class includes common fields for tracking creation and modification dates using @CreatedDate and @LastModifiedDate annotations. Both the Employee and Department entities extend this base class, inheriting its auditing capabilities. This setup ensures that whenever an entity is created or modified, the relevant timestamps are automatically updated, providing a comprehensive audit trail.

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

// Employee Projection Interface-Based

package com.example.yourapp.projection;

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

public interface EmployeeProjection {

Long getId();

String getFirstName();

String getLastName();

// Example of using @Value for custom projection

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

String getFullName();

}

// Department Projection Interface-Based

package com.example.yourapp.projection;

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

public interface DepartmentProjection {

Long getId();

String getName();

// Example of using @Value for custom projection

@Value("#{target.name + ' - ' + target.location}")

String getDepartmentDetails();

}

// Employee Projection Class-Based

package com.example.yourapp.projection;

public class EmployeeDTO {

private Long id;

private String firstName;

private String lastName;

private String fullName;

public EmployeeDTO(Long id, String firstName, String lastName) {

this.id = id;

this.firstName = firstName;

this.lastName = lastName;

this.fullName = firstName + " " + lastName;

}

// Getters and setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getFullName() {

return fullName;

}

public void setFullName(String fullName) {

this.fullName = fullName;

}}

// Department Projection Class-Based

package com.example.yourapp.projection;

public class DepartmentDTO {

private Long id;

private String name;

private String departmentDetails;

public DepartmentDTO(Long id, String name, String location) {

this.id = id;

this.name = name;

this.departmentDetails = name + " - " + location;

}

// Getters and setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDepartmentDetails() {

return departmentDetails;

}

public void setDepartmentDetails(String departmentDetails) {

this.departmentDetails = departmentDetails;

}}

### Explanation

**Interface-Based Projections**: Define projections using interfaces. Methods in these interfaces correspond to fields you want to fetch. Annotations like @Value can be used to customize the projection, such as concatenating fields to create a derived property like getFullName() or getDepartmentDetails()

**Class-Based Projections**: Define projections using classes with constructors. This approach allows more complex projections and customization. The constructor initializes fields and can create derived properties directly in the class, such as fullName or departmentDetails.

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

// PrimaryDataSourceConfig.java

package com.example.yourapp.config;

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

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;

@Configuration

public class PrimaryDataSourceConfig {

@Bean

@ConfigurationProperties("spring.datasource.primary")

public DataSourceProperties primaryDataSourceProperties() {

return new DataSourceProperties();

}

@Bean

@ConfigurationProperties("spring.datasource.primary.hikari")

public DataSource primaryDataSource() {

return primaryDataSourceProperties().initializeDataSourceBuilder()

.type(DriverManagerDataSource.class).build();

}

}

// SecondaryDataSourceConfig.java

package com.example.yourapp.config;

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

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;

@Configuration

public class SecondaryDataSourceConfig {

@Bean

@ConfigurationProperties("spring.datasource.secondary")

public DataSourceProperties secondaryDataSourceProperties() {

return new DataSourceProperties();

}

@Bean

@ConfigurationProperties("spring.datasource.secondary.hikari")

public DataSource secondaryDataSource() {

return secondaryDataSourceProperties().initializeDataSourceBuilder()

.type(DriverManagerDataSource.class).build();

}

}

### **Explanation**

### ****Spring Boot Auto-Configuration****: Utilize Spring Boot’s auto-configuration to set up data sources. Define data source properties and beans in separate configuration classes (PrimaryDataSourceConfig and SecondaryDataSourceConfig) for each data source.**Externalizing Configuration**: Store your data source configurations in application.properties. This includes database URLs, credentials, and other properties. This approach helps manage configurations easily and separates code from configuration.**Managing Multiple Data Sources**: By defining multiple configuration classes and properties, you can handle multiple data sources within the same application. Each data source is configured separately, allowing you to interact with different databases as needed.

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

// Employee.java

package com.example.yourapp.model;

import org.hibernate.annotations.DynamicUpdate;

import org.hibernate.annotations.Where;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

@DynamicUpdate // Automatically generates SQL updates only for changed fields

@Where(clause = "active = true") // Filter for soft deletes or specific conditions

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String firstName;

private String lastName;

private String email;

private boolean active = true; // Example of soft delete

// Getters and setters

}

# application.properties

# Hibernate Dialect

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

# Hibernate Show SQL

spring.jpa.show-sql=true

# Hibernate Format SQL

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

# Hibernate Batch Size

spring.jpa.properties.hibernate.jdbc.batch\_size=30

# Hibernate Enable Second Level Cache

spring.jpa.properties.hibernate.cache.use\_second\_level\_cache=true

// EmployeeService.java

package com.example.yourapp.service;

import com.example.yourapp.model.Employee;

import com.example.yourapp.repository.EmployeeRepository;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void batchSaveEmployees(List<Employee> employees) {

int batchSize = 30; // Example batch size

for (int i = 0; i < employees.size(); i++) {

employeeRepository.save(employees.get(i));

if (i % batchSize == 0 && i > 0) {

// Flush and clear the session to prevent memory issues

employeeRepository.flush();

employeeRepository.clear();

}}}}

### Explanation

**Hibernate-Specific Annotations**:@DynamicUpdate: Ensures that only modified fields are updated in the database, improving performance for frequent updates.@Where: Adds a filter condition to queries for entities, useful for soft deletes or specific data subsets.

**Configuring Hibernate Dialect and Properties**:Configure Hibernate dialect and other properties in application.properties to match your database and optimize performance. Properties like hibernate.dialect, hibernate.jdbc.batch\_size, and cache settings help in fine-tuning.

**Batch Processing**:Implement batch processing to handle bulk operations efficiently. By saving entities in batches and periodically flushing the session, you can avoid performance issues and reduce memory consumption.