**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.

Hibernate-Specific Annotations

import org.hibernate.annotations.CreationTimestamp;

import org.hibernate.annotations.UpdateTimestamp;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.ManyToOne;

import javax.persistence.Table;

import javax.persistence.Column;

import java.time.LocalDateTime;

@Entity

@Table(name = "employees")

public class Employee {

@Id

private Long id;

@Column(nullable = false, length = 100)

private String name;

@Column(nullable = false)

private String position;

@Column(nullable = false)

private Double salary;

@ManyToOne

private Department department;

@CreationTimestamp

@Column(updatable = false)

private LocalDateTime createdDate;

@UpdateTimestamp

private LocalDateTime updatedDate;

// Getters and Setters

}

Department Entity

import org.hibernate.annotations.BatchSize;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

import javax.persistence.Column;

import java.util.List;

@Entity

@Table(name = "departments")

@BatchSize(size = 20)

public class Department {

@Id

private Long id;

@Column(nullable = false, length = 100)

private String name;

// List of employees in the department

@OneToMany(mappedBy = "department")

private List<Employee> employees;

// Getters and Setters

}

**Configuring Hibernate Dialect and Properties**

# Hibernate Dialect

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

# Hibernate Properties

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

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

spring.jpa.properties.hibernate.jdbc.batch\_size=50

spring.jpa.properties.hibernate.order\_inserts=true

spring.jpa.properties.hibernate.order\_updates=true

# Show SQL

spring.jpa.show-sql=true

**Batch Processing**

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Repository;

import org.springframework.transaction.annotation.Transactional;

@Repository

public class EmployeeRepositoryCustomImpl {

@PersistenceContext

private EntityManager entityManager;

@Transactional

public void batchInsertEmployees(List<Employee> employees) {

int batchSize = 50;

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

entityManager.persist(employees.get(i));

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

// Flush a batch of inserts and release memory

entityManager.flush();

entityManager.clear();

}

}

// Ensure the last batch is processed

entityManager.flush();

entityManager.clear();

}

}

**Using Spring Data JPA for Batch Processing**

You can also leverage Spring Data JPA's built-in support for batch processing by configuring @Modifying queries.

**Repository Interface**

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

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

import org.springframework.data.repository.CrudRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface EmployeeRepository extends CrudRepository<Employee, Long> {

@Modifying

@Query("UPDATE Employee e SET e.salary = :salary WHERE e.position = :position")

void updateSalariesByPosition(Double salary, String position);

}