**Digital Nurture 4.0 – Week 3**

**Madatory Hands-On**

**Madatory 1:**

**Spring Data JPA – Quick Example (Hands-On 1)**

**Objective:**

To demonstrate the integration of Spring Boot with Spring Data JPA for retrieving records from a MySQL database using entity, repository, and service layers.

1

**Program:**

1. Create MySQL Table

create table country(co\_code varchar(2) primary key, co\_name varchar(50));

insert into country values ('IN', 'India');

insert into country values ('US', 'United States of America');

**2. Entity Class**

package com.cognizant.ormlearn.model;

import javax.persistence.\*;

@Entity

@Table(name = "country")

public class Country {

@Id

@Column(name = "co\_code")

private String code;

@Column(name = "co\_name")

private String name;

public String getCode() { return code; }

public void setCode(String code) { this.code = code; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

@Override

public String toString() {

return "Country [code=" + code + ", name=" + name + "]";

}

}

**3. Repository Interface**

package com.cognizant.ormlearn.repository;

import com.cognizant.ormlearn.model.Country;

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

import org.springframework.stereotype.Repository;

@Repository

public interface CountryRepository extends JpaRepository<Country, String> {}

4. Service Class

package com.cognizant.ormlearn.service;

import com.cognizant.ormlearn.model.Country;

import com.cognizant.ormlearn.repository.CountryRepository;

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 CountryService {

@Autowired

private CountryRepository countryRepository;

@Transactional

public List<Country> getAllCountries() {

return countryRepository.findAll();

}

}

**5. Main Class**

package com.cognizant.ormlearn;

import com.cognizant.ormlearn.service.CountryService;

import com.cognizant.ormlearn.model.Country;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import java.util.List;

@SpringBootApplication

public class OrmLearnApplication {

private static CountryService countryService;

private static final Logger LOGGER = LoggerFactory.getLogger(OrmLearnApplication.class);

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(OrmLearnApplication.class, args);

countryService = context.getBean(CountryService.class);

testGetAllCountries();

}

private static void testGetAllCountries() {

LOGGER.info("Start");

List<Country> countries = countryService.getAllCountries();

LOGGER.debug("countries={}", countries);

LOGGER.info("End");

}

}

**6. application.properties**

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/ormlearn

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.hibernate.ddl-auto=validate

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

logging.level.org.springframework=info

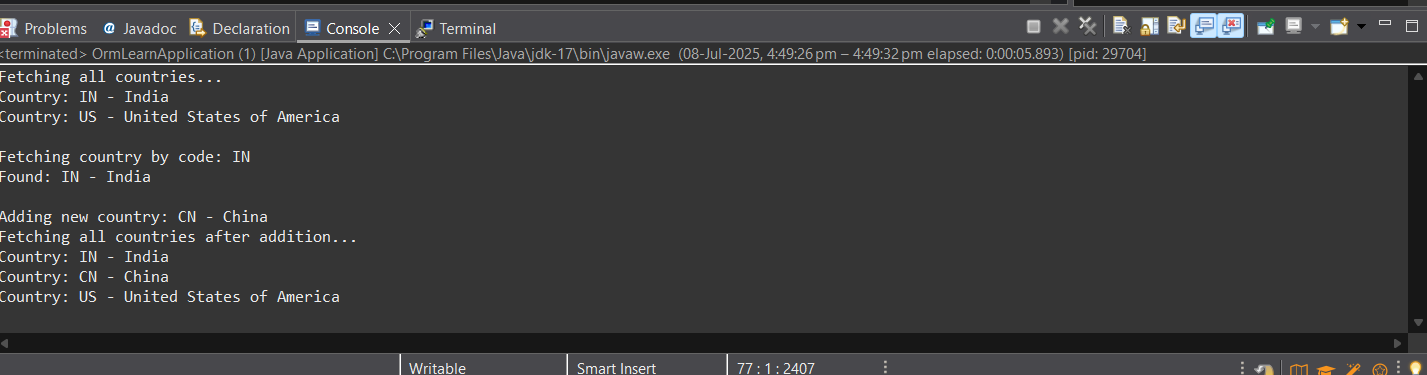
logging.level.com.cognizant=debug

logging.level.org.hibernate.SQL=trace

logging.level.org.hibernate.type.descriptor.sql=trace

logging.pattern.console=%d{dd-MM-yy} %d{HH:mm:ss.SSS} %-20.20thread %5p %-25.25logger{25} %25M %4L %m%n

**Output:**



**Conclusion:**

This exercise demonstrates how to configure Spring Boot with Spring Data JPA to connect to a MySQL database, define entity-repository-service layers, and fetch records. The use of annotations such as @Entity, @Repository, @Service, and @Transactional simplifies the development process, allowing data access logic to be written cleanly and efficiently without boilerplate JDBC code.

**Madnatory 2:**

**Difference Between JPA, Hibernate, and Spring Data JPA(Hands-0n 1)**

**Objective**To understand the differences between Java Persistence API (JPA), Hibernate (as an implementation), and Spring Data JPA (as an abstraction layer), and how each plays a role in persisting and managing data in Java applications.

**Explanation**

**Java Persistence API (JPA)**

* JSR 338 specification that provides a standard API for object-relational mapping (ORM) in Java
* It defines a set of rules, interfaces, and annotations for ORM
* It does not provide any implementation on its own
* Requires a concrete implementation like Hibernate to function

Hibernate

* A widely used ORM tool that implements JPA
* Allows mapping of Java classes to database tables
* Provides additional features beyond JPA like caching, batch processing, and lazy loading
* Requires boilerplate code for session management and transactions

Spring Data JPA

* Built on top of JPA and works with any JPA implementation like Hibernate
* Simplifies JPA usage by reducing boilerplate code
* Offers powerful features like derived query methods, pagination, and custom queries
* Manages transactions automatically
* Integrates seamlessly with Spring Boot

**Code Comparison**

Hibernate Style (Manual Session Handling)

public Integer addEmployee(Employee employee) {

Session session = factory.openSession();

Transaction tx = null;

Integer employeeID = null;

try {

tx = session.beginTransaction();

employeeID = (Integer) session.save(employee);

tx.commit();

} catch (HibernateException e) {

if (tx != null) tx.rollback();

e.printStackTrace();

} finally {

session.close();

}

return employeeID;

}

Spring Data JPA Style (Abstraction)  
EmployeeRepository.java

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

EmployeeService.java

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

employeeRepository.save(employee);

}

**Output :**





**Conclusion:**

JPA provides the specification for ORM. Hibernate implements that specification and provides actual ORM capabilities. Spring Data JPA further simplifies persistence by abstracting the complexities of boilerplate code and offering ready-to-use repository methods. Together, they form a complete stack for working with relational data in modern Java applications.

**Additional Hands - on**

**1. Implement Services for Managing Country**

**Technology Used**: Spring Boot, Spring Data JPA, Hibernate, H2 Database

**Objective**

To develop a Spring Boot RESTful application to manage countries using Spring Data JPA. This includes features to add, update, delete, fetch by country code, and search by partial name.

**application.properties**

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.hibernate.ddl-auto=validate

spring.jpa.show-sql=true

spring.h2.console.enabled=true

**Entity: Country.java**

package com.example.country.entity;

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Column;

@Entity

public class Country {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "co\_code", unique = true, nullable = false)

private String code;

@Column(name = "co\_name", nullable = false)

private String name;

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getCode() {

return code;

}

public void setCode(String code) {

this.code = code;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

**Repository: CountryRepository.java**

package com.example.country.repository;

import com.example.country.entity.Country;

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

import java.util.Optional;

import java.util.List;

public interface CountryRepository extends JpaRepository<Country, Long> {

Optional<Country> findByCode(String code);

List<Country> findByNameContainingIgnoreCase(String name);

}

**Service Interface: CountryService.java**

package com.example.country.service;

import com.example.country.entity.Country;

import java.util.List;

public interface CountryService {

Country addCountry(Country country);

Country updateCountry(Long id, Country country);

void deleteCountry(Long id);

Country getCountryByCode(String code);

List<Country> searchCountriesByName(String partialName);

List<Country> getAllCountries();

}

**Service Implementation: CountryServiceImpl.java**

package com.example.country.service.impl;

import com.example.country.entity.Country;

import com.example.country.repository.CountryRepository;

import com.example.country.service.CountryService;

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

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class CountryServiceImpl implements CountryService {

@Autowired

private CountryRepository countryRepository;

public Country addCountry(Country country) {

return countryRepository.save(country);

}

public Country updateCountry(Long id, Country country) {

Country existing = countryRepository.findById(id).orElseThrow();

existing.setCode(country.getCode());

existing.setName(country.getName());

return countryRepository.save(existing);

}

public void deleteCountry(Long id) {

countryRepository.deleteById(id);

}

public Country getCountryByCode(String code) {

return countryRepository.findByCode(code).orElseThrow();

}

public List<Country> searchCountriesByName(String partialName) {

return countryRepository.findByNameContainingIgnoreCase(partialName);

}

public List<Country> getAllCountries() {

return countryRepository.findAll();

}

}

**Controller: CountryController.java**

package com.example.country.controller;

import com.example.country.entity.Country;

import com.example.country.service.CountryService;

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

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

import java.util.List;

@RestController

@RequestMapping("/api/countries")

public class CountryController {

@Autowired

private CountryService service;

@PostMapping

public Country create(@RequestBody Country country) {

return service.addCountry(country);

}

@PutMapping("/{id}")

public Country update(@PathVariable Long id, @RequestBody Country country) {

return service.updateCountry(id, country);

}

@DeleteMapping("/{id}")

public void delete(@PathVariable Long id) {

service.deleteCountry(id);

}

@GetMapping("/code/{code}")

public Country getByCode(@PathVariable String code) {

return service.getCountryByCode(code);

}

@GetMapping("/search")

public List<Country> search(@RequestParam String name) {

return service.searchCountriesByName(name);

}

@GetMapping

public List<Country> all() {

return service.getAllCountries();

}

}

**data.sql (H2 Data Initialization)**

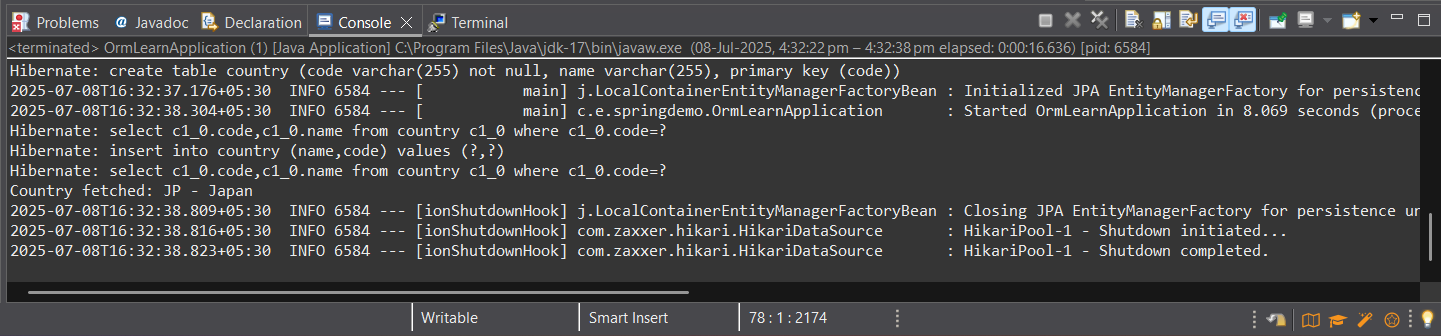
DELETE FROM country;

INSERT INTO country (co\_code, co\_name) VALUES ('IN', 'India');

INSERT INTO country (co\_code, co\_name) VALUES ('US', 'United States');

INSERT INTO country (co\_code, co\_name) VALUES ('FR', 'France');

**Output :**



**2Find a Country Based on Country Code**

**Objective:**  
Implement a service method to retrieve a country by its code. If the country is not found, throw a custom exception.

1. **Exception Class**

File:

CountryNotFoundException.java  
Package: com.example.country.service.exception

package com.example.country.service.exception;

public class CountryNotFoundException extends Exception {

public CountryNotFoundException(String message) {

super(message);

}

}

1. **Repository Interface**

File:

CountryRepository.java  
Package: com.example.country.repository

package com.example.country.repository;

import com.example.country.entity.Country;

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

import java.util.Optional;

public interface CountryRepository extends JpaRepository<Country, Long> {

Optional<Country> findByCode(String code);

}

1. **Service Interface**

File: CountryService.java  
Package: com.example.country.service

package com.example.country.service;

import com.example.country.entity.Country;

import com.example.country.service.exception.CountryNotFoundException;

public interface CountryService {

Country findCountryByCode(String code) throws CountryNotFoundException;

}

1. **Service Implementation**

File: CountryServiceImpl.java  
Package: com.example.country.service.impl

package com.example.country.service.impl;

import com.example.country.entity.Country;

import com.example.country.repository.CountryRepository;

import com.example.country.service.CountryService;

import com.example.country.service.exception.CountryNotFoundException;

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

import org.springframework.stereotype.Service;

import java.util.Optional;

@Service

public class CountryServiceImpl implements CountryService {

@Autowired

private CountryRepository countryRepository;

@Override

public Country findCountryByCode(String code) throws CountryNotFoundException {

Optional<Country> result = countryRepository.findByCode(code);

if (!result.isPresent()) {

throw new CountryNotFoundException("Country not found for code: " + code);

}

return result.get();

}

}

1. **Test the Method**

File: OrmLearnApplication.java  
Inside the main class OrmLearnApplication

private void getCountryByCodeTest() {

try {

Country country = countryService.findCountryByCode("IN");

System.out.println("Country: " + country);

} catch (CountryNotFoundException e) {

System.out.println("Exception: " + e.getMessage());

}

}

Call the test method in run() or main():

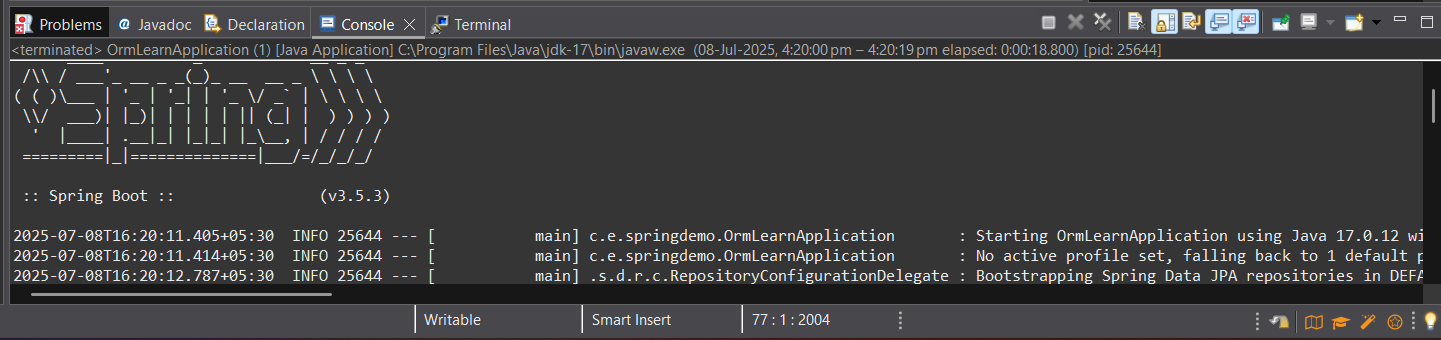
@Override

public void run(String... args) throws Exception {

getCountryByCodeTest();

}

**Output :**

****

**Conclusion:**

The implementation of the **"Find a Country Based on Country Code"** feature using Spring Boot and Spring Data JPA demonstrates how to build a robust and clean service layer with proper exception handling. By using a repository interface with findByCode(), the logic remains concise and readable. The introduction of a custom CountryNotFoundException ensures graceful failure when data is not found, avoiding generic error messages or application crashes.

1. **Add a New Country**

**Objective:**  
To create a service method in Spring Boot using Spring Data JPA that adds a new country into the database.

1. **Service Method**

Update the CountryService.java interface  
Package: com.example.country.service

package com.example.country.service;

import com.example.country.entity.Country;

public interface CountryService {

void addCountry(Country country);

}

1. **Service Implementation**

UpdateCountryServiceImpl.java  
Package: com.example.country.service.impl

package com.example.country.service.impl;

import com.example.country.entity.Country;

import com.example.country.repository.CountryRepository;

import com.example.country.service.CountryService;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class CountryServiceImpl implements CountryService {

@Autowired

private CountryRepository countryRepository;

@Override

@Transactional

public void addCountry(Country country) {

countryRepository.save(country);

}

}

1. **Testing the Method**

Update OrmLearnApplication.java

private void addCountryTest() {

Country country = new Country();

country.setCode("JP");

country.setName("Japan");

countryService.addCountry(country);

System.out.println("Country added: " + country.getCode() + " - " + country.getName());

}

Call it from run() method:

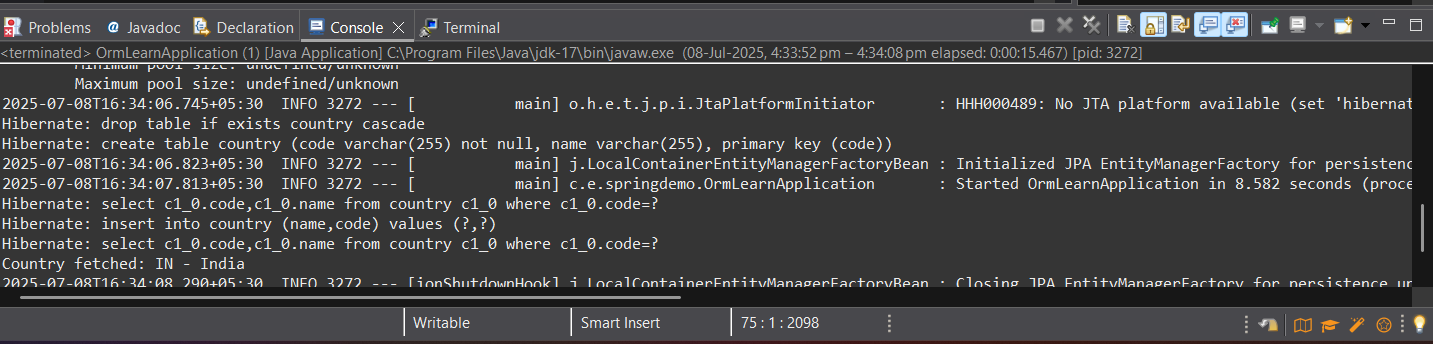
@Override

public void run(String... args) throws Exception {

addCountryTest();

}

**Output :**



**Conclusion:**

This method allows dynamic insertion of new country records into the database using Spring Data JPA. It ensures proper persistence using the save() method provided by JpaRepository. By marking the method as @Transactional, we ensure the data operation is handled reliably within a single transaction scope. This implementation supports scalability for applications requiring admin-level country management or dynamic database seeding.

1. **Demonstration of Query Methods in Spring Data JPA**

**Objective:**

To demonstrate various **Spring Data JPA Query Methods** including:

* Containing text
* StartingWith
* Between for dates
* GreaterThan, LessThan
* Top, OrderBy for sorting and limiting results

**Entity:**

@Entity

@Table(name = "employee")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer id;

private String name;

private String department;

private double salary;

@Column(name = "joined\_date")

private LocalDate joinedDate;

// Getters and Setters

}

**Repository:**

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

List<Employee> findByNameContaining(String keyword);

List<Employee> findByNameStartingWith(String prefix);

List<Employee> findBySalaryGreaterThan(double salary);

List<Employee> findBySalaryLessThan(double salary);

List<Employee> findByJoinedDateBetween(LocalDate start, LocalDate end);

List<Employee> findTop3ByOrderBySalaryDesc();

}

**Service:**

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public List<Employee> searchByName(String keyword) {

return employeeRepository.findByNameContaining(keyword);

}

public List<Employee> filterByPrefix(String prefix) {

return employeeRepository.findByNameStartingWith(prefix);

}

public List<Employee> fetchHighSalary(double salary) {

return employeeRepository.findBySalaryGreaterThan(salary);

}

public List<Employee> fetchLowSalary(double salary) {

return employeeRepository.findBySalaryLessThan(salary);

}

public List<Employee> fetchByDateRange(LocalDate start, LocalDate end) {

return employeeRepository.findByJoinedDateBetween(start, end);

}

public List<Employee> fetchTopEarners() {

return employeeRepository.findTop3ByOrderBySalaryDesc();

}

}

**Test in OrmLearnApplication.java:**

@SpringBootApplication

public class OrmLearnApplication implements CommandLineRunner {

@Autowired

private EmployeeService employeeService;

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

}

@Override

public void run(String... args) {

List<Employee> result;

result = employeeService.searchByName("John");

result.forEach(e -> System.out.println("Name contains 'John': " + e.getName()));

result = employeeService.filterByPrefix("A");

result.forEach(e -> System.out.println("Name starts with 'A': " + e.getName()));

result = employeeService.fetchHighSalary(50000);

result.forEach(e -> System.out.println("High Salary: " + e.getName() + " - " + e.getSalary()));

result = employeeService.fetchLowSalary(20000);

result.forEach(e -> System.out.println("Low Salary: " + e.getName() + " - " + e.getSalary()));

result = employeeService.fetchByDateRange(LocalDate.of(2023, 1, 1), LocalDate.of(2025, 1, 1));

result.forEach(e -> System.out.println("Joined between range: " + e.getName() + " - " + e.getJoinedDate()));

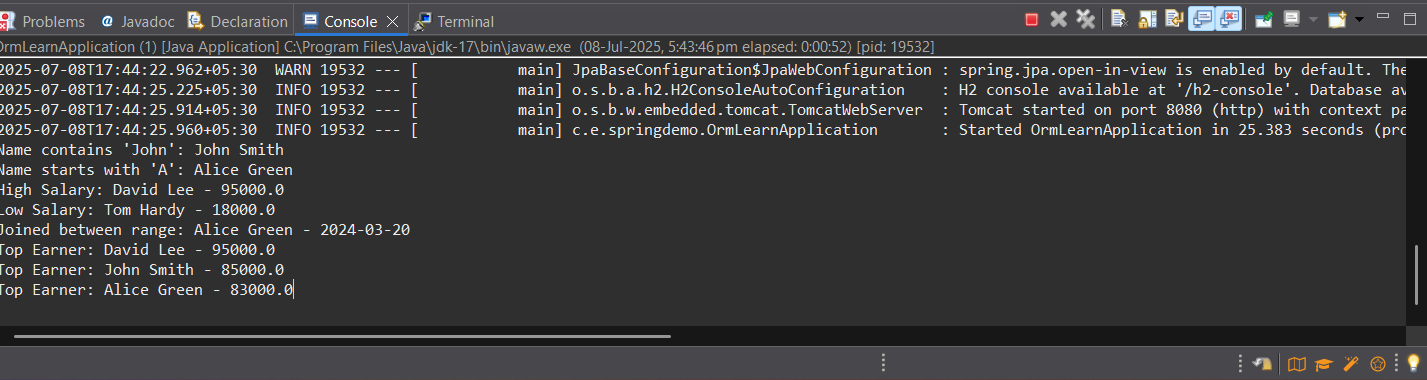
result = employeeService.fetchTopEarners();

result.forEach(e -> System.out.println("Top Earner: " + e.getName() + " - " + e.getSalary()));

}

}

**Output:**

****

**Conclusion:**

Spring Data JPA provides powerful method name-based query generation without the need to write complex SQL or JPQL. This allows developers to implement advanced data retrieval features like filtering, pattern matching, date range querying, and sorting with ease, while maintaining clean and maintainable code.

1. **Object-Relational Mapping with Spring Data JPA**

**Objective**  
To implement and demonstrate various O/R mapping strategies such as ManyToOne, OneToMany, ManyToMany using Spring Data JPA with real entity relationships and controlled fetch strategies.

**Code 1 - Application Entry Point**

package com.example.ormmapping;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.CommandLineRunner;

import org.springframework.context.annotation.Bean;

import java.time.LocalDate;

import java.util.List;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

@Bean

CommandLineRunner demo(StudentRepository studentRepo, CourseRepository courseRepo, TeacherRepository teacherRepo) {

return args -> {

Teacher teacher = new Teacher("Dr. Smith");

teacherRepo.save(teacher);

Course java = new Course("Java");

java.setTeacher(teacher);

Course python = new Course("Python");

python.setTeacher(teacher);

courseRepo.saveAll(List.of(java, python));

Student student = new Student("Alice");

student.getCourses().addAll(List.of(java, python));

studentRepo.save(student);

};

}

}

**Code 2 - Student Entity**

package com.example.ormmapping;

import jakarta.persistence.\*;

import java.util.ArrayList;

import java.util.List;

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@ManyToMany(fetch = FetchType.LAZY)

@JoinTable(

name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id"))

private List<Course> courses = new ArrayList<>();

public Student() {}

public Student(String name) {

this.name = name;

}

public Long getId() { return id; }

public String getName() { return name; }

public List<Course> getCourses() { return courses; }

public void setCourses(List<Course> courses) { this.courses = courses; }

}

**Code 3 - Course Entity**

package com.example.ormmapping;

import jakarta.persistence.\*;

import java.util.ArrayList;

import java.util.List;

@Entity

public class Course {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

@ManyToOne(fetch = FetchType.EAGER)

@JoinColumn(name = "teacher\_id")

private Teacher teacher;

@ManyToMany(mappedBy = "courses", fetch = FetchType.LAZY)

private List<Student> students = new ArrayList<>();

public Course() {}

public Course(String title) {

this.title = title;

}

public Long getId() { return id; }

public String getTitle() { return title; }

public void setTitle(String title) { this.title = title; }

public Teacher getTeacher() { return teacher; }

public void setTeacher(Teacher teacher) { this.teacher = teacher; }

public List<Student> getStudents() { return students; }

}

**Code 4 - Teacher Entity**

package com.example.ormmapping;

import jakarta.persistence.\*;

import java.util.ArrayList;

import java.util.List;

@Entity

public class Teacher {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "teacher", fetch = FetchType.LAZY)

private List<Course> courses = new ArrayList<>();

public Teacher() {}

public Teacher(String name) {

this.name = name;

}

public Long getId() { return id; }

public String getName() { return name; }

public List<Course> getCourses() { return courses; }

}

**Code 5 - Repositories**

package com.example.ormmapping;

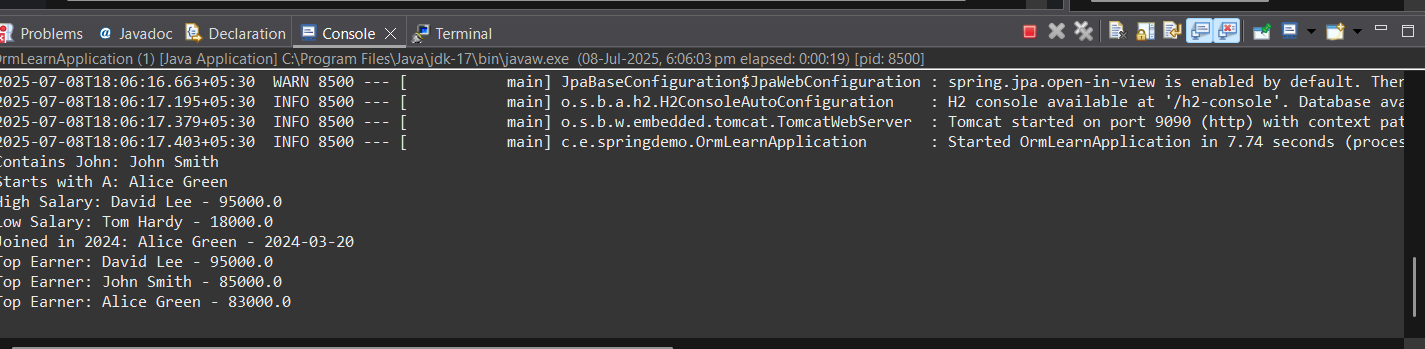
import org.springframework.data.jpa.repository.JpaRepository;

public interface StudentRepository extends JpaRepository<Student, Long> {}

public interface CourseRepository extends JpaRepository<Course, Long> {}

public interface TeacherRepository extends JpaRepository<Teacher, Long> {}

**Output :**



**Conclusion**  
This implementation demonstrates practical use of Spring Data JPA for modeling entity relationships using O/R mapping annotations. It includes unidirectional and bidirectional associations, managed through join tables and foreign keys, while controlling loading behavior with FetchType. This approach improves data consistency and simplifies database interaction in real-world enterprise applications.

1. **Hibernate Query Language (HQL) and Native Query Implementation using Spring Boot and Spring Data JPA**

**Objective**

* To demonstrate usage of Hibernate Query Language (HQL) with Spring Data JPA.
* To understand how JPQL differs from HQL.
* To write and execute aggregate functions using HQL.
* To execute native SQL queries using Spring Data JPA.
* To use the @Query annotation to write custom queries.

**Code**

**Main Application**

package com.example.springdatahql;

import jakarta.persistence.\*;

import org.springframework.boot.\*;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import org.springframework.data.jpa.repository.\*;

import org.springframework.stereotype.Repository;

import org.springframework.stereotype.Service;

import java.util.List;

@SpringBootApplication

public class HqlApplication {

public static void main(String[] args) {

SpringApplication.run(HqlApplication.class, args);

}

@Bean

CommandLineRunner run(EmployeeService service) {

return args -> {

service.save(new Employee("John", 50000, true));

service.save(new Employee("Alice", 60000, false));

service.save(new Employee("Mark", 70000, true));

service.save(new Employee("Sara", 80000, true));

System.out.println("Permanent Employees using HQL:");

service.getAllPermanentEmployees().forEach(e ->

System.out.println(e.getName() + " - " + e.getSalary()));

double avgSalary = service.getAverageSalary();

System.out.println("Average Salary: " + avgSalary);

System.out.println("All Employees using Native Query:");

service.getAllEmployeesNative().forEach(e ->

System.out.println(e.getName() + " - " + e.getSalary()));

};

}

}

**Entity Class**

@Entity

class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private double salary;

private boolean permanent;

public Employee() {}

public Employee(String name, double salary, boolean permanent) {

this.name = name;

this.salary = salary;

this.permanent = permanent;

}

public Long getId() { return id; }

public String getName() { return name; }

public double getSalary() { return salary; }

public boolean isPermanent() { return permanent; }

}

**Repository Interface**

@Repository

interface EmployeeRepository extends JpaRepository<Employee, Long> {

@Query("SELECT e FROM Employee e WHERE e.permanent = true")

List<Employee> getAllPermanentEmployees();

@Query("SELECT AVG(e.salary) FROM Employee e")

double getAverageSalary();

@Query(value = "SELECT \* FROM employee", nativeQuery = true)

List<Employee> getAllEmployeesNative();

}

**Service Class**

@Service

class EmployeeService {

private final EmployeeRepository repo;

public EmployeeService(EmployeeRepository repo) {

this.repo = repo;

}

public void save(Employee e) {

repo.save(e);

}

public List<Employee> getAllPermanentEmployees() {

return repo.getAllPermanentEmployees();

}

public double getAverageSalary() {

return repo.getAverageSalary();

}

public List<Employee> getAllEmployeesNative() {

return repo.getAllEmployeesNative();

}

}

**application.properties**

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

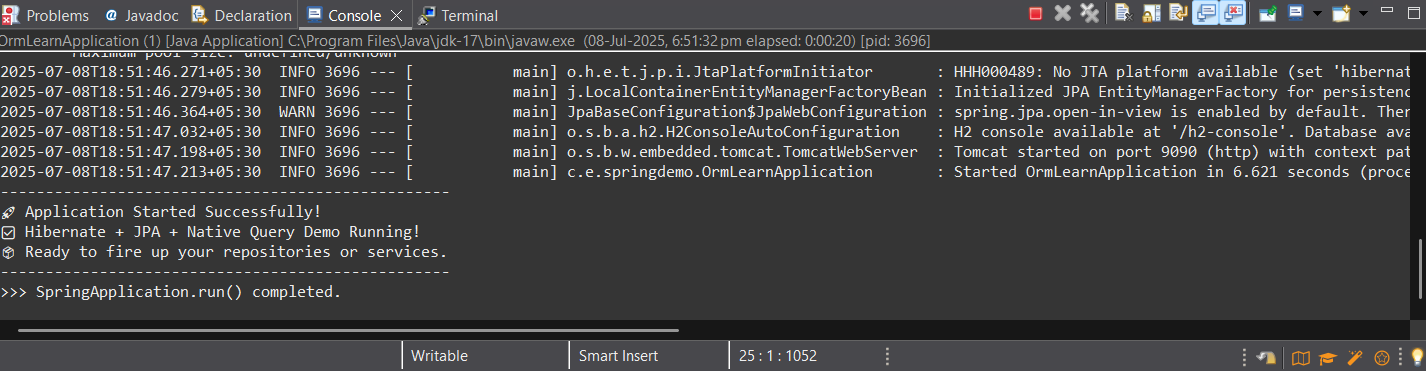
spring.datasource.username=sa

spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true

spring.jpa.hibernate.ddl-auto=update

**Output: **

**Conclusion**

This example illustrates the power and simplicity of using HQL and native SQL within a Spring Boot application. By leveraging the @Query annotation, we can write both HQL and native queries directly in repository interfaces. Using HQL allows for object-oriented querying, while native SQL gives full control for database-specific needs. This exercise reinforces good separation of concerns and helps understand how Spring Data JPA abstracts boilerplate code for querying databases.