**WEEK 3-MANDATORY HANDS ON EXERCISES**

**Module 5 - Spring Core and Maven**

**✅ Exercise 1: Configuring a Basic Spring Application**

**What I did**

* I created a **Maven project** named LibraryManagement.
* In the pom.xml file, I added Spring Core dependencies to support the basic Spring features.
* I created an XML configuration file called applicationContext.xml inside src/main/resources.
* In this XML file, I defined beans for BookService and BookRepository.
* Then, I created the BookService class inside the com.library.service package and the BookRepository class inside the com.library.repository package.
* Finally, I created a main class (LibraryManagementApplication) to load the Spring context and test whether everything was configured properly.

**✅ Exercise 2: Implementing Dependency Injection**

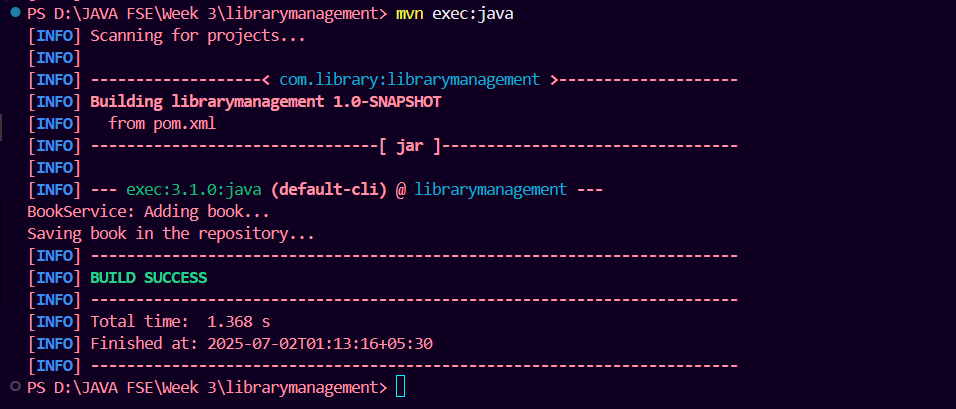
**What I did**

* I updated the applicationContext.xml file to **wire BookRepository into BookService** using Spring's dependency injection.
* In the BookService class, I added a **setter method** for BookRepository so that Spring can inject the dependency using setter injection.
* I tested this setup by running the main application class to confirm that BookService could successfully use BookRepository. The dependency injection worked as expected.

**✅ Exercise 4: Creating and Configuring a Maven Project**

**What I did**

* I created a new **Maven project** named LibraryManagement to manage dependencies and build the project easily.
* In the pom.xml, I added dependencies for **Spring Context**, **Spring AOP**, and **Spring WebMVC** to support different Spring features that may be used in the project.
* I also configured the Maven Compiler Plugin in the pom.xml to set the Java version to 1.8, which ensures compatibility during compilation.

**Output:**

**Module 6 - Spring Data JPA with Spring Boot, Hibernate**

**✅ Hands-on 1: Spring Data JPA — Quick Example**

**📄 Documentation**

In this hands-on exercise, I worked on creating a demo project using Spring Data JPA with Hibernate as the JPA implementation and connected it to a MySQL database. Below I have described in detail what I did step by step**.**

**✨ Overview**

In this project, I learned how to set up a Spring Boot application with Spring Data JPA, configure MySQL, and perform basic operations on a Country table. This helped me understand how Spring Boot simplifies data access using JPA and Hibernate.

**✅ Software Prerequisites**

* MySQL Server 8.0
* MySQL Workbench 8
* Eclipse IDE for Enterprise Java Developers (2019-03 R)
* Maven 3.6.2

**✅ Project Setup**

**What I did**

* I generated the project using Spring Initializr (<https://start.spring.io/>):
  + Group: com.cognizant
  + Artifact: orm-learn
  + Description: "Demo project for Spring Data JPA and Hibernate"
  + Selected dependencies: Spring Boot DevTools, Spring Data JPA, and MySQL Driver
* Downloaded and extracted the zip file into my Eclipse workspace.
* Imported it into Eclipse using File > Import > Maven > Existing Maven Projects.

**✅ Database Setup**

**What I did**

* Opened MySQL client and created a schema named ormlearn:

create schema ormlearn;

* Created a country table and inserted records:

sql

CopyEdit

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');

**✅ Configuration**

**What I did**

* Updated src/main/resources/application.properties with database connection details and logging configurations.

Example 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

* Built the project using Maven to verify everything is working.

**✅ Model Layer**

**What I did**

* Created package: com.cognizant.ormlearn.model.
* Created a class Country with fields for code and name.
* Annotated the class with @Entity and @Table, and annotated fields with @Column and @Id.

Example snippet:

@Entity

@Table(name = "country")

public class Country {

@Id

@Column(name = "co\_code")

private String code;

@Column(name = "co\_name")

private String name;

// getters, setters, toString()

}

**✅ Repository Layer**

**What I did**

* Created package: com.cognizant.ormlearn.repository.
* Created CountryRepository interface that extends JpaRepository<Country, String>.
* Added @Repository annotation.

**✅ Service Layer**

**What I did**

* Created package: com.cognizant.ormlearn.service.
* Created class CountryService with @Service annotation.
* Autowired CountryRepository.
* Created method getAllCountries() with @Transactional, which calls countryRepository.findAll() and returns the list of countries.

✅ Application Class & Testing

What I did

* In OrmLearnApplication.java, added SLF4J logger for logging messages.
* Used ApplicationContext to fetch CountryService bean.
* Called getAllCountries() method inside a test method to verify data retrieval.
* Checked logs to confirm data from the country table was fetched successfully.

Example code snippet:

private static void testGetAllCountries() {

LOGGER.info("Start");

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

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

LOGGER.info("End");

}

**✅ Additional Concepts Learned**

JPA

* JPA (Java Persistence API) is a specification for mapping Java objects to relational database tables.
* It does not provide its own implementation but defines standard interfaces.

Hibernate

* Hibernate is an ORM framework that implements JPA specification.
* It helps in automatically mapping Java classes to database tables.

Spring Data JPA

* Spring Data JPA is an abstraction over JPA/Hibernate.
* It reduces boilerplate code and manages repository creation, transaction handling, and query generation automatically.

**✅ SME Walkthrough Points**

**What I explored**

* src/main/java: Contains application source code.
* src/main/resources: Contains configuration files such as application.properties.
* src/test/java: Contains test code.
* OrmLearnApplication.java: Main entry point where I tested the application and verified service method calls.
* @SpringBootApplication: This annotation combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.
* pom.xml: Verified dependencies, plugins, and used Dependency Hierarchy to understand transitive dependencies.

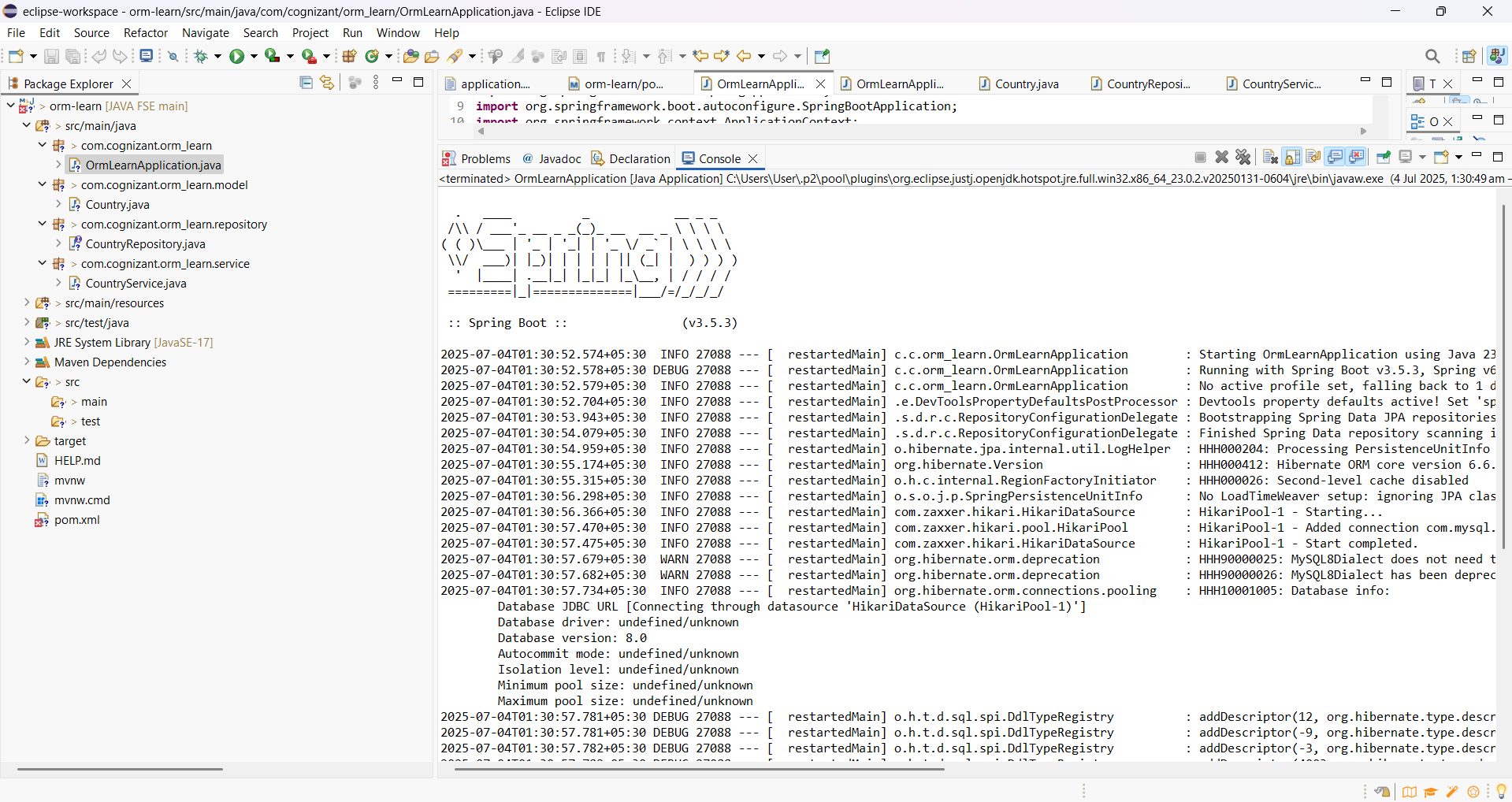
**💬 Conclusion**

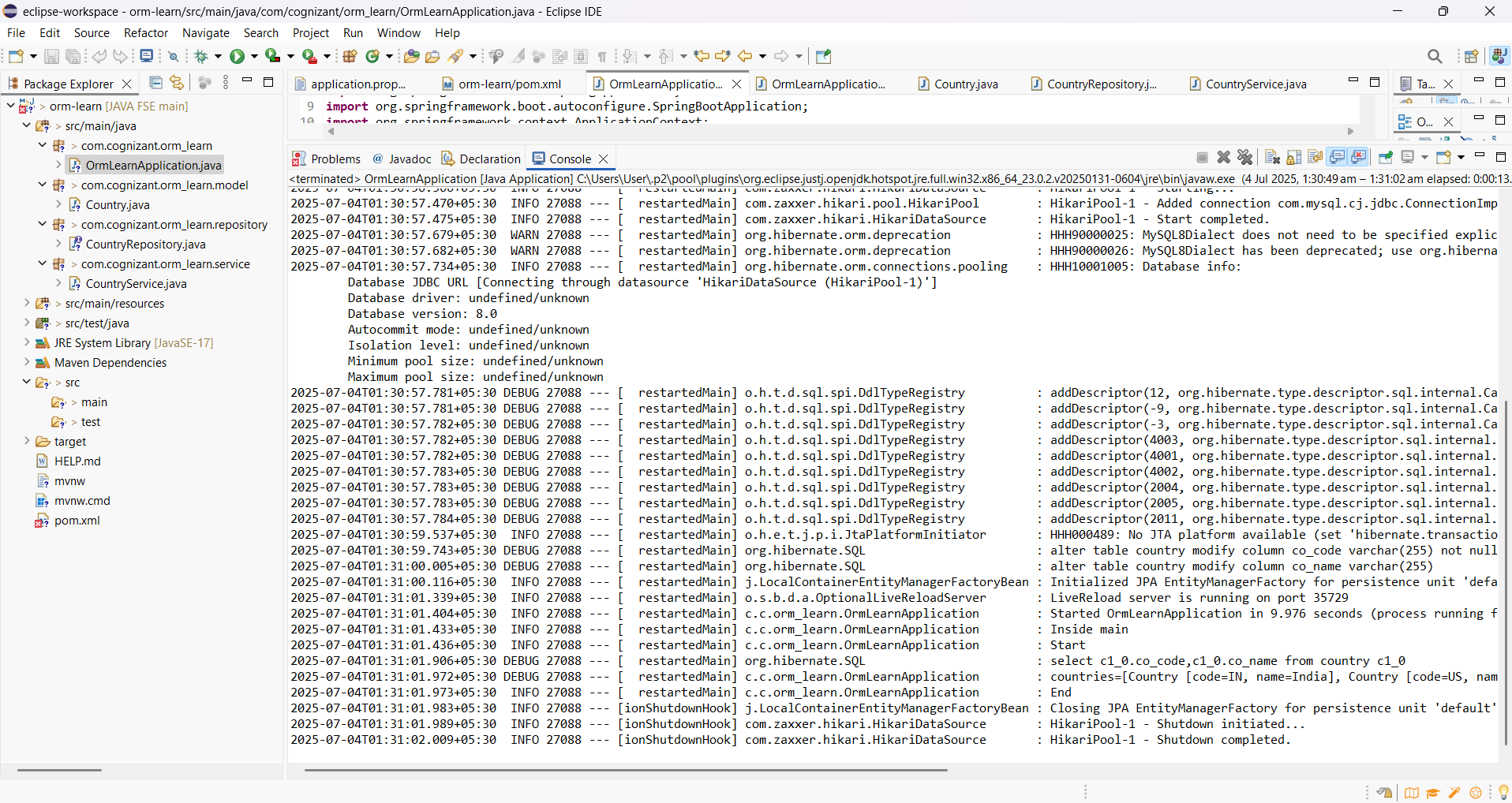
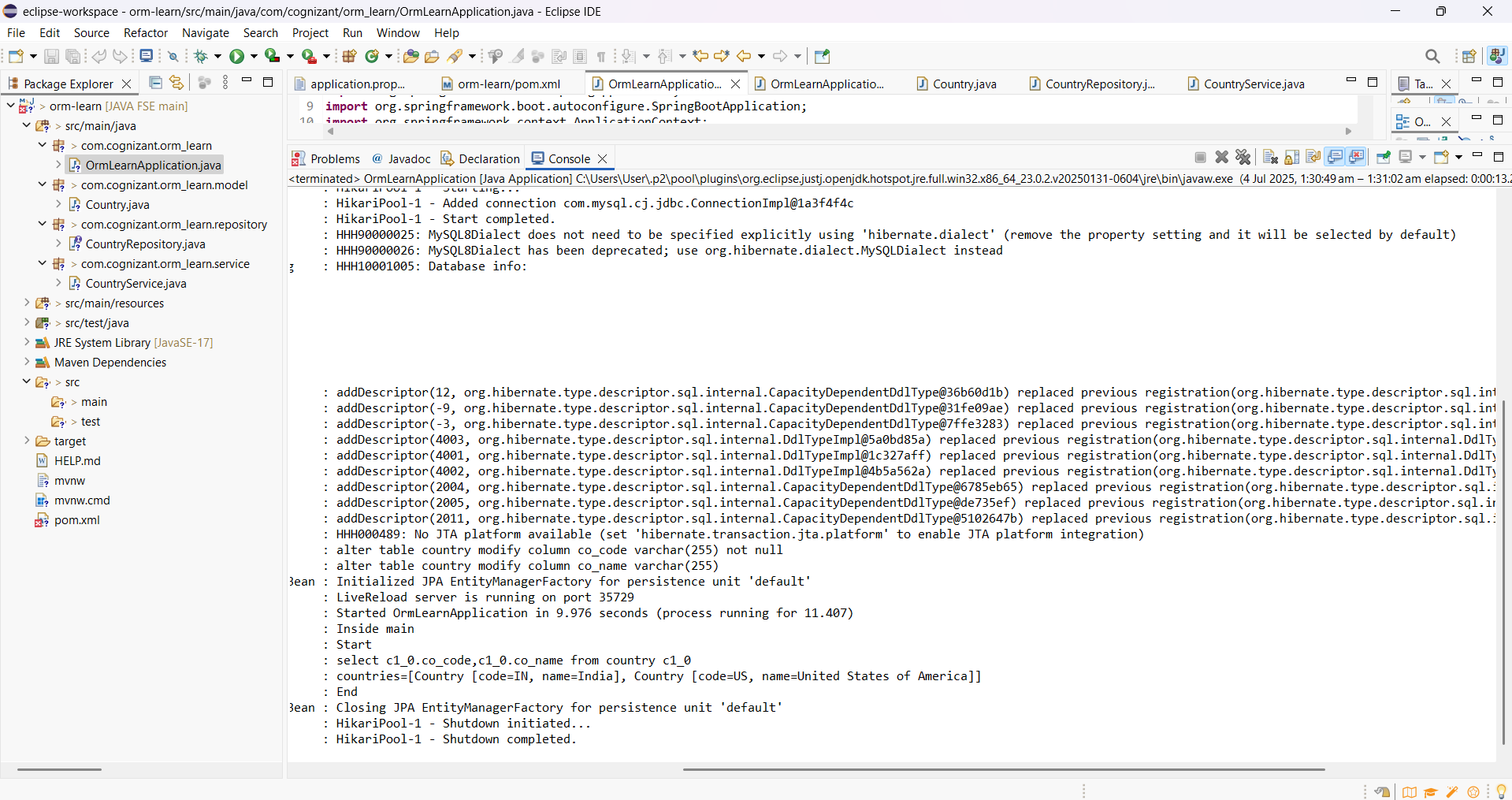
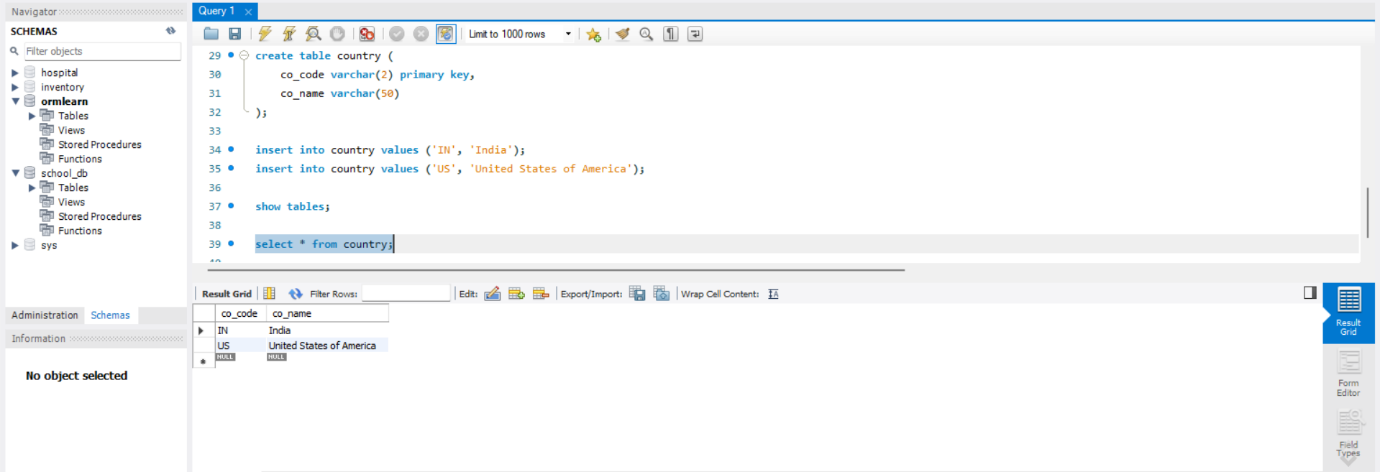
**Through this hands-on exercise, I successfully:**

* Set up a Spring Boot application with Spring Data JPA and Hibernate.
* Connected the application to a MySQL database.
* Created and mapped entity classes to database tables.
* Implemented repository and service layers using Spring Data JPA.
* Verified database interactions by running and checking logs.

**I have completed, tested, and pushed all my code to my GitHub repository.**

**Output:**

****

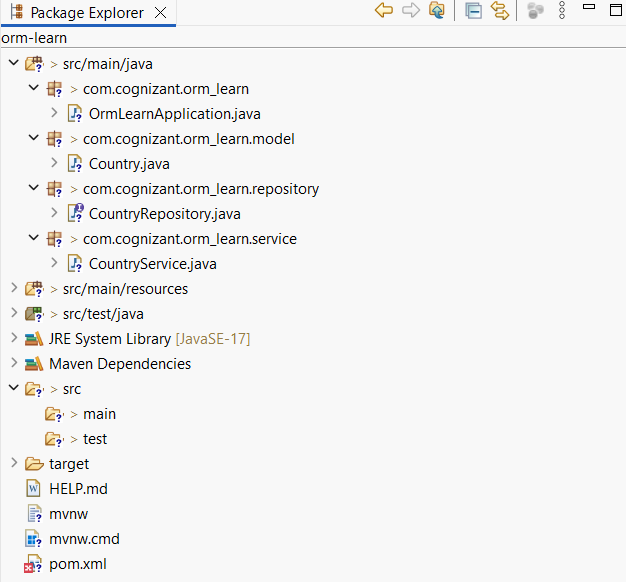
****

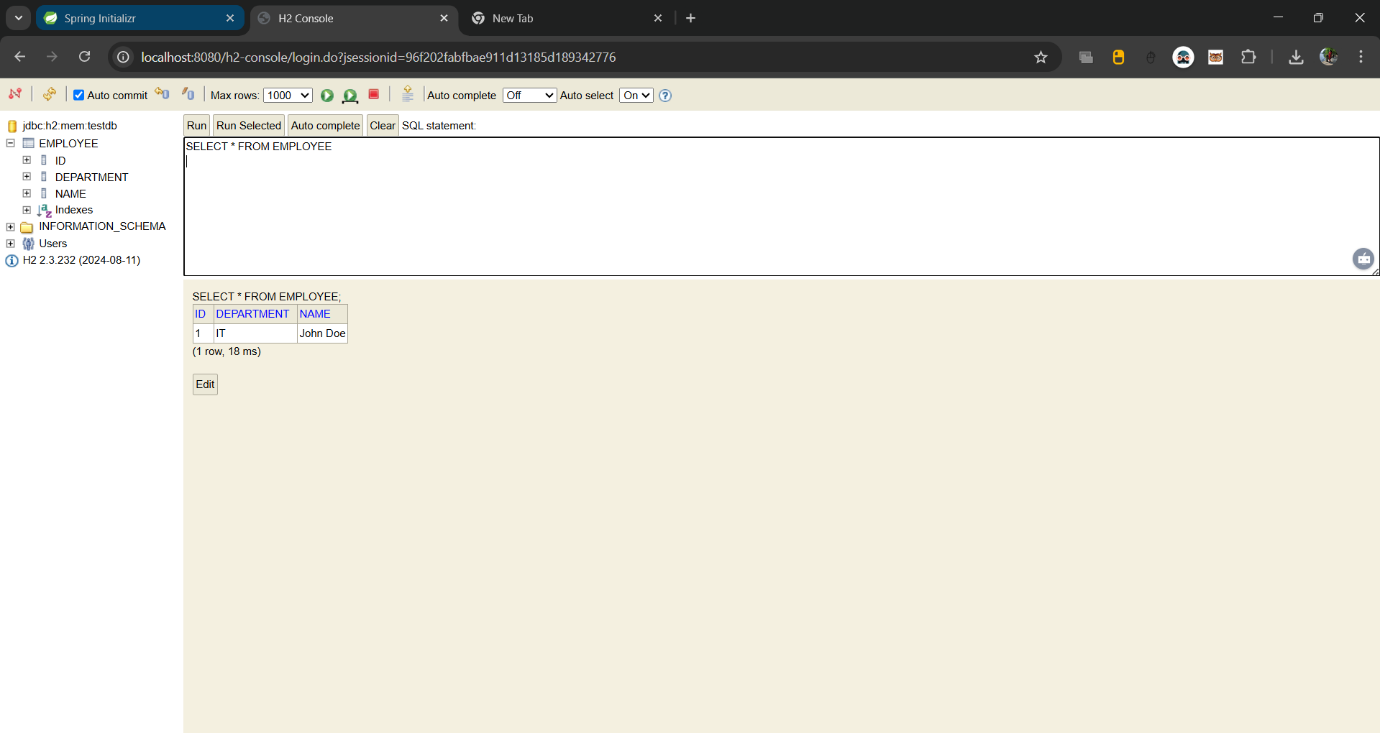
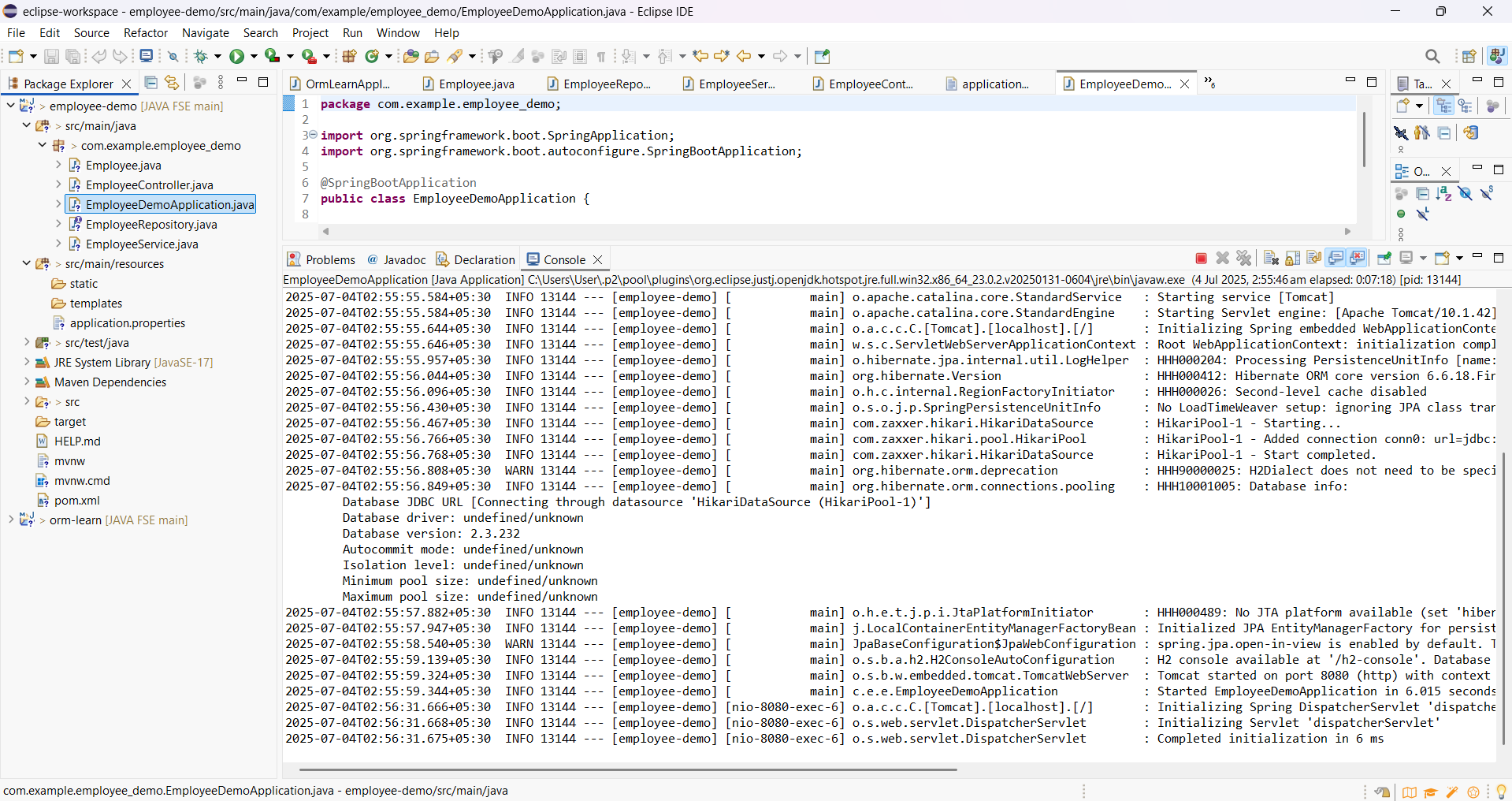
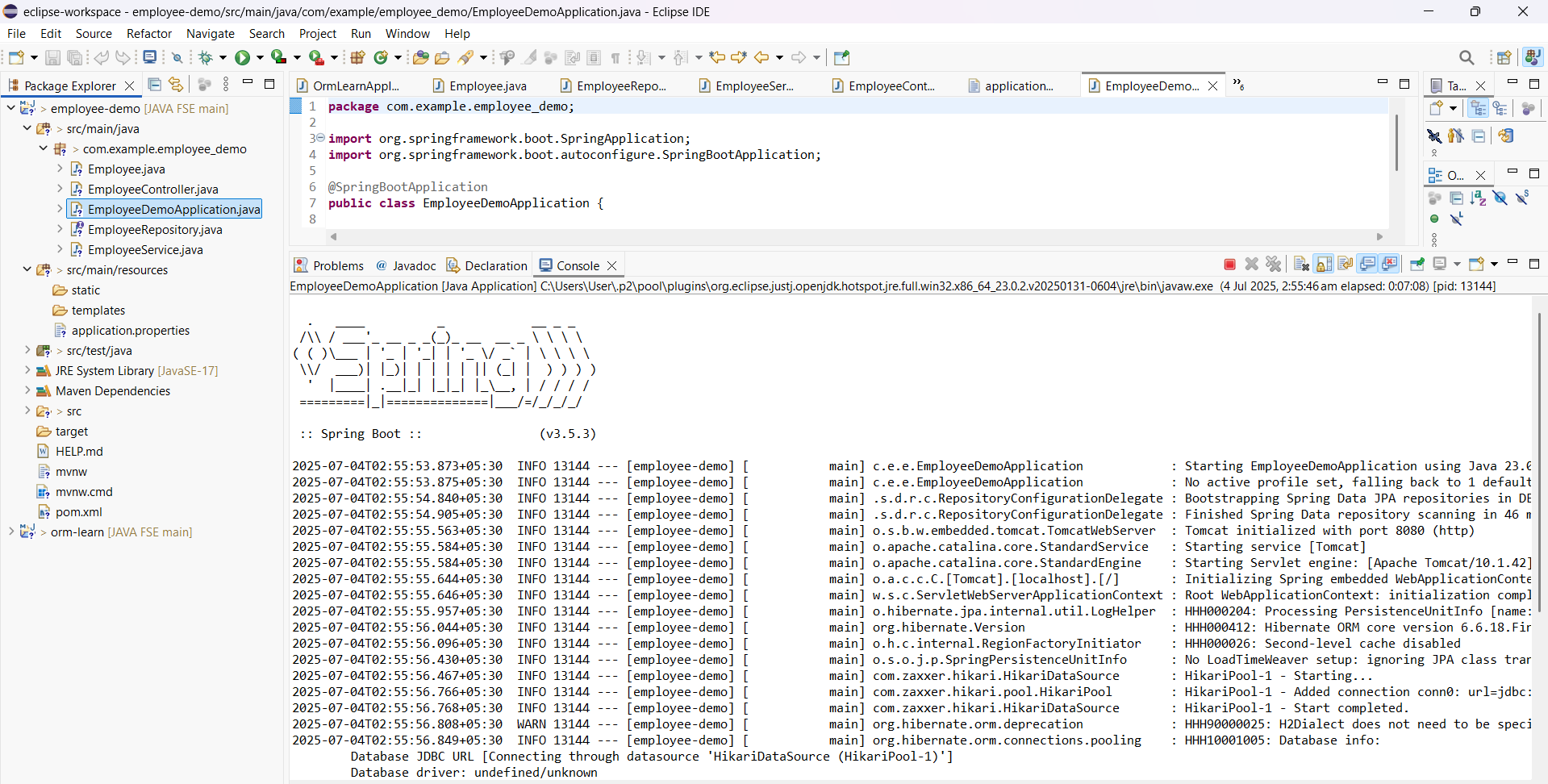
**✅ Hands-on 4: Difference between JPA, Hibernate, and Spring Data JPA**

**📄 Differences in Tabular Form**

|  |  |  |  |
| --- | --- | --- | --- |
| **Aspect** | **JPA** | **Hibernate** | **Spring Data JPA** |
| Type | Specification (standard) | Implementation of JPA | Abstraction layer over JPA implementations |
| Provides | Interfaces & guidelines only | Concrete ORM features & APIs | Simplified repository APIs, auto queries |
| Implementation | No own implementation | Provides actual implementation | Depends on JPA provider (like Hibernate) |
| Boilerplate | Medium | Less than plain JDBC but still manual | Minimal, handles most CRUD & transaction code |
| Transaction Mgmt | Not directly handled | Manually managed (session, transaction) | Managed automatically by Spring framework |
| Query Support | Supports JPQL | Supports HQL & JPQL | Supports JPQL, auto query creation from method names |

**✅ What I did**

* I implemented the employee example using both Hibernate and Spring Data JPA to clearly see the difference.
* In Hibernate, I had to write manual session and transaction handling code.
* In Spring Data JPA, I created an EmployeeRepository interface extending JpaRepository, and used it inside a service class with @Autowired, which made the code much simpler and cleaner.



**WEEK 3-ADDITIONAL HANDS ON EXERCISES**

**🟢 Exercise 5: Configuring Spring IoC Container (XML)**

In this exercise, I configured Spring IoC container using XML.

1️⃣ I created an applicationContext.xml file inside src/main/resources.  
2️⃣ In this XML file, I defined beans for BookService and BookRepository, and injected BookRepository into BookService using a setter method.  
3️⃣ I added a setter method in the BookService class to enable this injection.  
4️⃣ I also created a main class to load the Spring context from the XML and call the addBook() method to test it.

👉 The output was almost the same as before (for example, BookService: Adding book...). Only the configuration style changed, not the logic.

**🟢 Exercise 7: Constructor and Setter Injection**

In this exercise, I explored using both constructor and setter injection in Spring.

1️⃣ I modified the applicationContext.xml to configure constructor injection for BookService using <constructor-arg ref="bookRepository"/>.  
2️⃣ I kept the setter method for BookRepository and configured it as needed using <property name="bookRepository" ref="bookRepository"/>.  
3️⃣ I tested this by running the same main class, which confirmed both types of injections worked properly.

👉 The output was the same as before. This exercise mainly helped me understand different ways of injecting dependencies in Spring.

**✅ Exercise 9: Creating a Spring Boot Application**

**🔹 My Objective**

In this exercise, I created a Spring Boot application for the library management system to simplify configuration, reduce boilerplate, and make deployment easier.

**🔹 Steps I Followed**

**1️⃣ Created Spring Boot Project**

I used **Spring Initializr** to generate a new project with the following details:

* Project: Maven
* Group: com.library
* Artifact & Name: LibraryManagementSpring
* Package: com.library.LibraryManagementSpring
* Java version: 17

**2️⃣ Added Required Dependencies**

I included:

* Spring Web
* Spring Data JPA
* H2 Database

These helped me build REST APIs, interact with the database easily using JPA, and use an in-memory database for testing.

**3️⃣ Configured application.properties**

I added configuration in src/main/resources/application.properties to set up the H2 database:

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

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

This allows me to use an in-memory database and access it through the H2 console.

**4️⃣ Defined Entity and Repository**

I created an **entity class** Book to represent the book data and a **repository interface** BookRepository to handle database operations.

**5️⃣ Created REST Controller**

I created BookController with CRUD endpoints to:

* Add a new book
* Get all books
* Get a book by ID
* Update a book
* Delete a book

**6️⃣ Ran and Tested the Application**

* I ran LibraryManagementSpringApplication.java.
* Console showed: Started LibraryManagementSpringApplication, meaning the server started successfully.
* I tested the endpoints using **Postman**:
  + **POST** /books to add a book.
  + **GET** /books to fetch all books.
  + **GET** /books/{id} to fetch a single book.
  + **PUT** /books/{id} to update book details.
  + **DELETE** /books/{id} to delete a book.
* I also checked the database visually using the H2 console at <http://localhost:8080/h2-console>.

**🔹 Output**

The application ran successfully, and all CRUD operations worked as expected.  
The data was correctly saved and retrieved from the H2 in-memory database, confirming the integration of Spring Boot, JPA, and REST APIs.

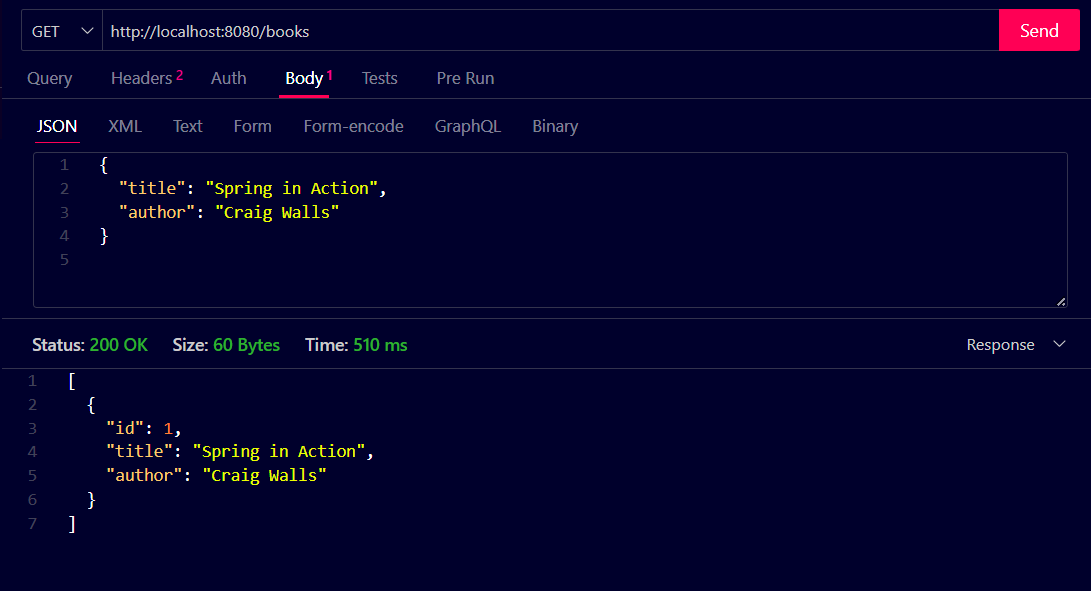
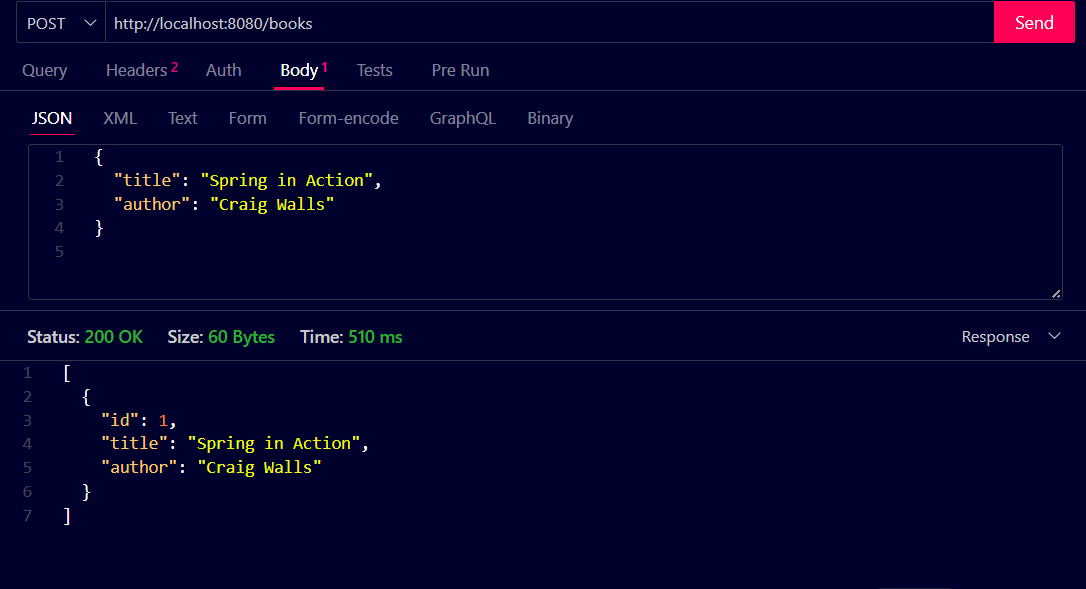
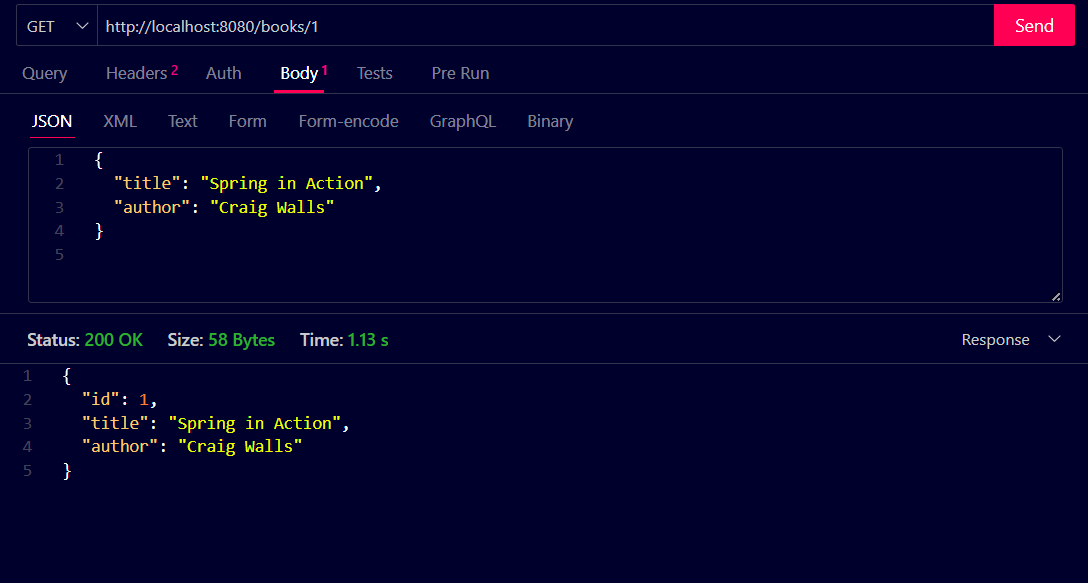
**✅ Conclusion**

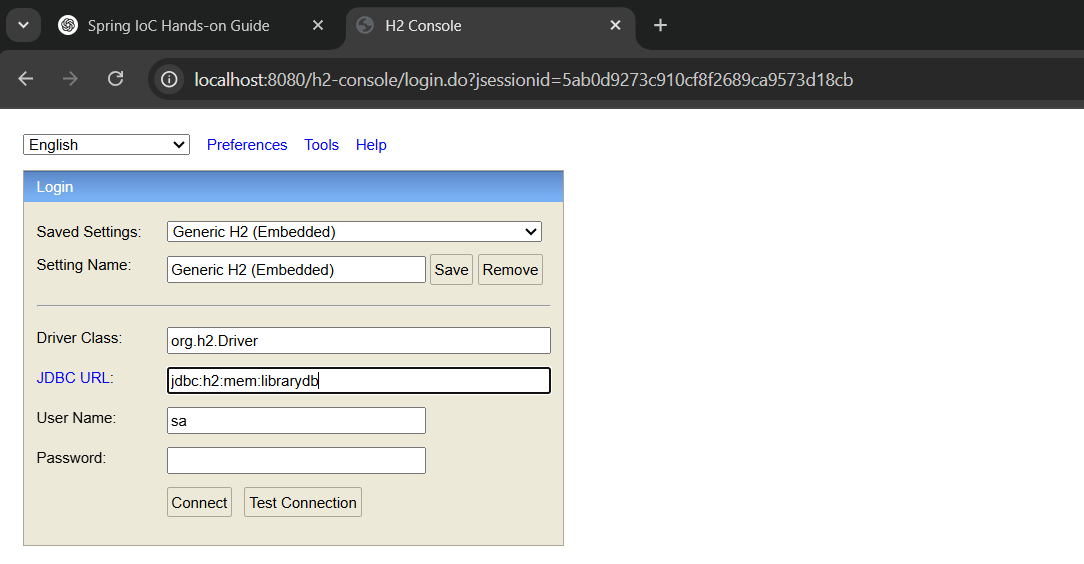
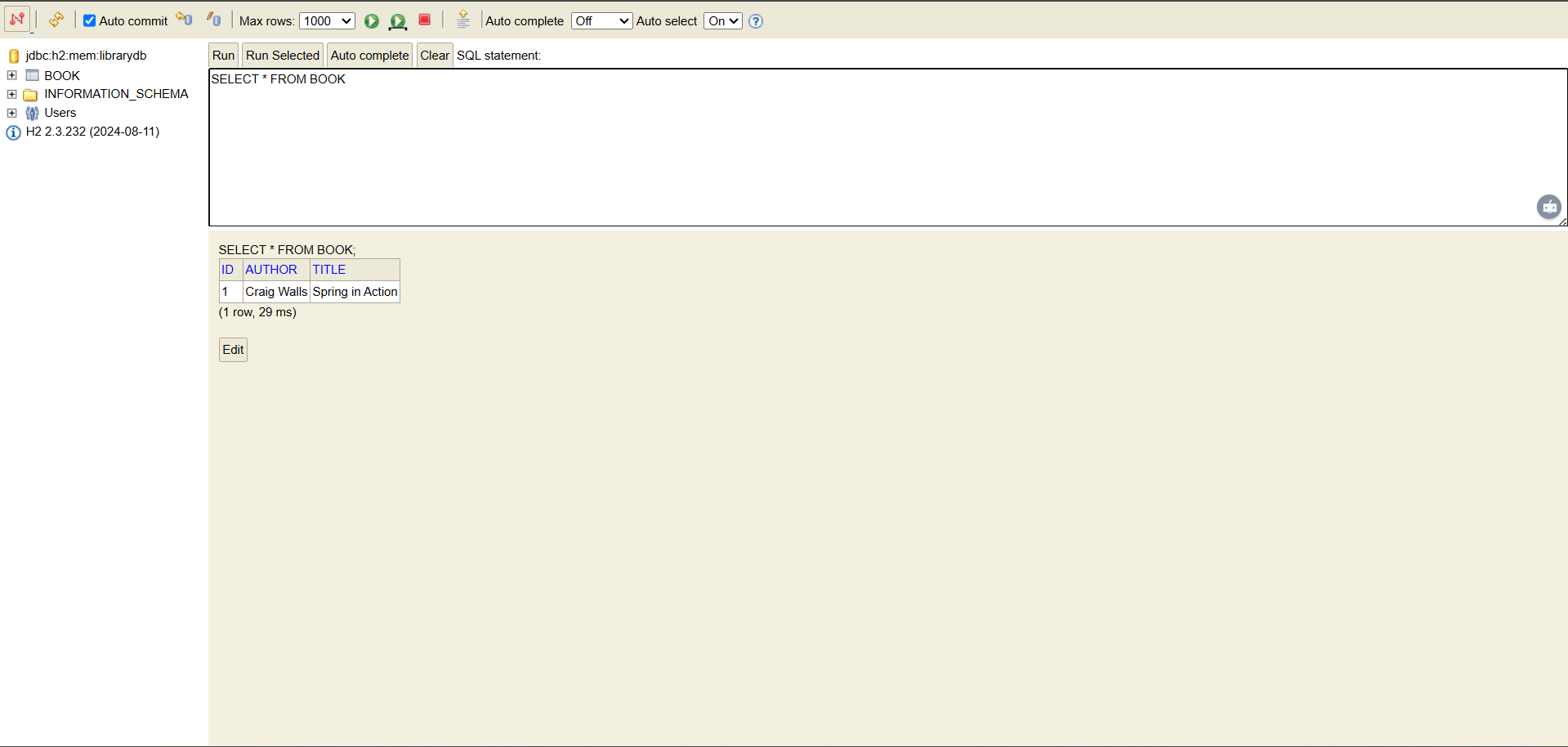
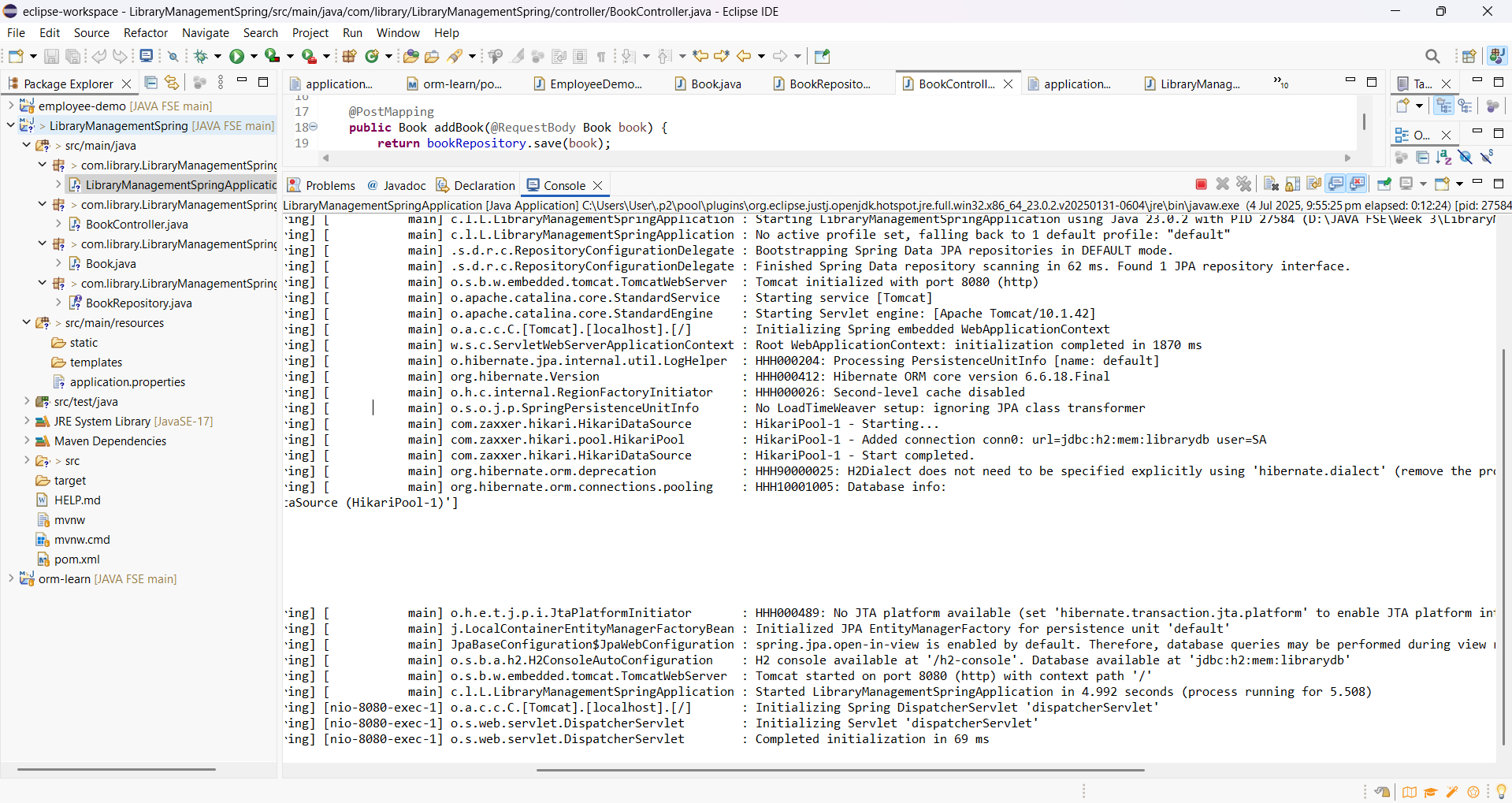
Through this exercise, I learned how to:

* Create a Spring Boot project from scratch
* Configure database connections using application.properties
* Build a fully working REST API with Spring Data JPA
* Use H2 console for quick data verification

This exercise simplified the overall setup and gave me a clear idea of building production-ready Spring Boot applications.

**Outputs:**

****

****

**Module 6 - Spring Data JPA with Spring Boot, Hibernate**

**1. spring-data-jpa-handson**

**Hands on 5: Country Management Service Using Spring Data JPA**

**Introduction**

This document outlines the step-by-step process I followed to implement services for managing countries in a Spring Boot application using Spring Data JPA. The major functionalities include:

* **Finding a country by country code----second addition hands on**
* **Adding a new country----third additional hands on**
* **Updating an existing country**
* **Deleting a country**
* **Searching countries by partial country name**

My project builds on the foundation completed in **Hands on 1** and leverages a MySQL database to persist country data.

**Project Setup and Configuration**

**1. Project Setup**

* Created a Spring Boot project using Spring Initializr with dependencies:
  + Spring Boot DevTools
  + Spring Data JPA
  + MySQL Driver
* Imported the project into Eclipse IDE for Enterprise Java Developers.

**2. Database Configuration (MySQL)**

* Configured **src/main/resources/application.properties**
* Added MySQL connector dependency to **pom.xml**:

1<dependency>

2 <groupId>mysql</groupId>

3 <artifactId>mysql-connector-java</artifactId>

4 <scope>runtime</scope>

5</dependency>

**Code Implementation**

**1. Country Entity**

**src/main/java/com/cognizant/orm\_learn/model/Country.java**

package com.cognizant.orm\_learn.model;

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

@Entity

@Table(name="country")

public class Country {

@Id

@Column(name="co\_code")

private String code;

@Column(name="co\_name")

private String name;

// Getters and setters

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;

}

// toString method for easy logging

@Override

public String toString() {

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

}

}

**2. Country Repository**

**src/main/java/com/cognizant/orm\_learn/repository/CountryRepository.java**

package com.cognizant.orm\_learn.repository;

import java.util.List;

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

import org.springframework.stereotype.Repository;

import com.cognizant.orm\_learn.model.Country;

@Repository

public interface CountryRepository extends JpaRepository<Country, String> {

List<Country> findByNameContainingIgnoreCase(String partialName);

}

This repository interface extends **JpaRepository** to get basic CRUD methods and adds a custom finder method for partial name matching.

**3. Country Service**

**src/main/java/com/cognizant/orm\_learn/service/CountryService.java**

package com.cognizant.orm\_learn.service;

import java.util.List;

import jakarta.transaction.Transactional;

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

import org.springframework.stereotype.Service;

import com.cognizant.orm\_learn.model.Country;

import com.cognizant.orm\_learn.repository.CountryRepository;

@Service

public class CountryService {

@Autowired

private CountryRepository countryRepository;

@Transactional

public List<Country> getAllCountries() {

return countryRepository.findAll();

}

@Transactional

public Country getCountryByCode(String code) {

return countryRepository.findById(code).orElse(null);

}

@Transactional

public Country addCountry(Country country) {

if (countryRepository.existsById(country.getCode())) {

throw new RuntimeException("Country with code " + country.getCode() + " already exists.");

}

return countryRepository.save(country);

}

@Transactional

public Country updateCountry(Country country) {

if (!countryRepository.existsById(country.getCode())) {

throw new RuntimeException("Country with code " + country.getCode() + " does not exist.");

}

return countryRepository.save(country);

}

@Transactional

public void deleteCountry(String code) {

if (!countryRepository.existsById(code)) {

throw new RuntimeException("Country with code " + code + " does not exist.");

}

countryRepository.deleteById(code);

}

@Transactional

public List<Country> findCountriesByPartialName(String partialName) {

return countryRepository.findByNameContainingIgnoreCase(partialName);

}

}

**Data Population**

Created a **data.sql** file inside **src/main/resources** with the following format to load large country dataset at startup:

SQL

1delete from country;

2

3insert into country (co\_code, co\_name) values ('AF', 'Afghanistan');

4insert into country (co\_code, co\_name) values ('AL', 'Albania');

5-- Insert all other countries similarly --

6insert into country (co\_code, co\_name) values ('AX', 'Åland Islands');

The application property **spring.jpa.hibernate.ddl-auto=update** ensures tables are created or updated automatically.

**Testing the Service**

I tested the service methods by adding code inside the Spring Boot main application class:

**src/main/java/com/cognizant/orm\_learn/OrmLearnApplication.java**

Java

Collapse

1package com.cognizant.orm\_learn;

2

3import com.cognizant.orm\_learn.model.Country;

4import com.cognizant.orm\_learn.service.CountryService;

5import org.springframework.boot.SpringApplication;

6import org.springframework.boot.autoconfigure.SpringBootApplication;

7import org.springframework.context.ApplicationContext;

8import java.util.List;

9

10@SpringBootApplication

11public class OrmLearnApplication {

12

13 private static CountryService countryService;

14

15 public static void main(String[] args) {

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

17 countryService = context.getBean(CountryService.class);

18

19 testGetCountryByCode();

20 testAddCountry();

21 testUpdateCountry();

22 testDeleteCountry();

23 testFindByPartialName();

24 }

25

26 private static void testGetCountryByCode() {

27 System.out.println("=== Get Country by Code 'IN' ===");

28 Country country = countryService.getCountryByCode("IN");

29 System.out.println(country);

30 }

31

32 private static void testAddCountry() {

33 System.out.println("=== Add Country 'XY' - 'Xylophone Land' ===");

34 Country newCountry = new Country();

35 newCountry.setCode("XY");

36 newCountry.setName("Xylophone Land");

37 try {

38 countryService.addCountry(newCountry);

39 System.out.println("Added: " + countryService.getCountryByCode("XY"));

40 } catch (Exception e) {

41 System.err.println("Error adding country: " + e.getMessage());

42 }

43 }

44

45 private static void testUpdateCountry() {

46 System.out.println("=== Update Country 'XY' name to 'Xylophone World' ===");

47 Country updatedCountry = new Country();

48 updatedCountry.setCode("XY");

49 updatedCountry.setName("Xylophone World");

50 try {

51 countryService.updateCountry(updatedCountry);

52 System.out.println("Updated: " + countryService.getCountryByCode("XY"));

53 } catch (Exception e) {

54 System.err.println("Error updating country: " + e.getMessage());

55 }

56 }

57

58 private static void testDeleteCountry() {

59 System.out.println("=== Delete Country with Code 'XY' ===");

60 try {

61 countryService.deleteCountry("XY");

62 System.out.println("Deleted country 'XY'");

63 } catch (Exception e) {

64 System.err.println("Error deleting country: " + e.getMessage());

65 }

66 }

67

68 private static void testFindByPartialName() {

69 System.out.println("=== Find countries containing 'an' ===");

70 List<Country> countries = countryService.findCountriesByPartialName("an");

71 countries.forEach(System.out::println);

72 }

73}

Running this main method confirmed:

* Fetching a country by code
* Adding a new country
* Updating an existing country’s name
* Deleting a country
* Searching by partial country name

All operations worked successfully, with SQL queries displayed in the console.

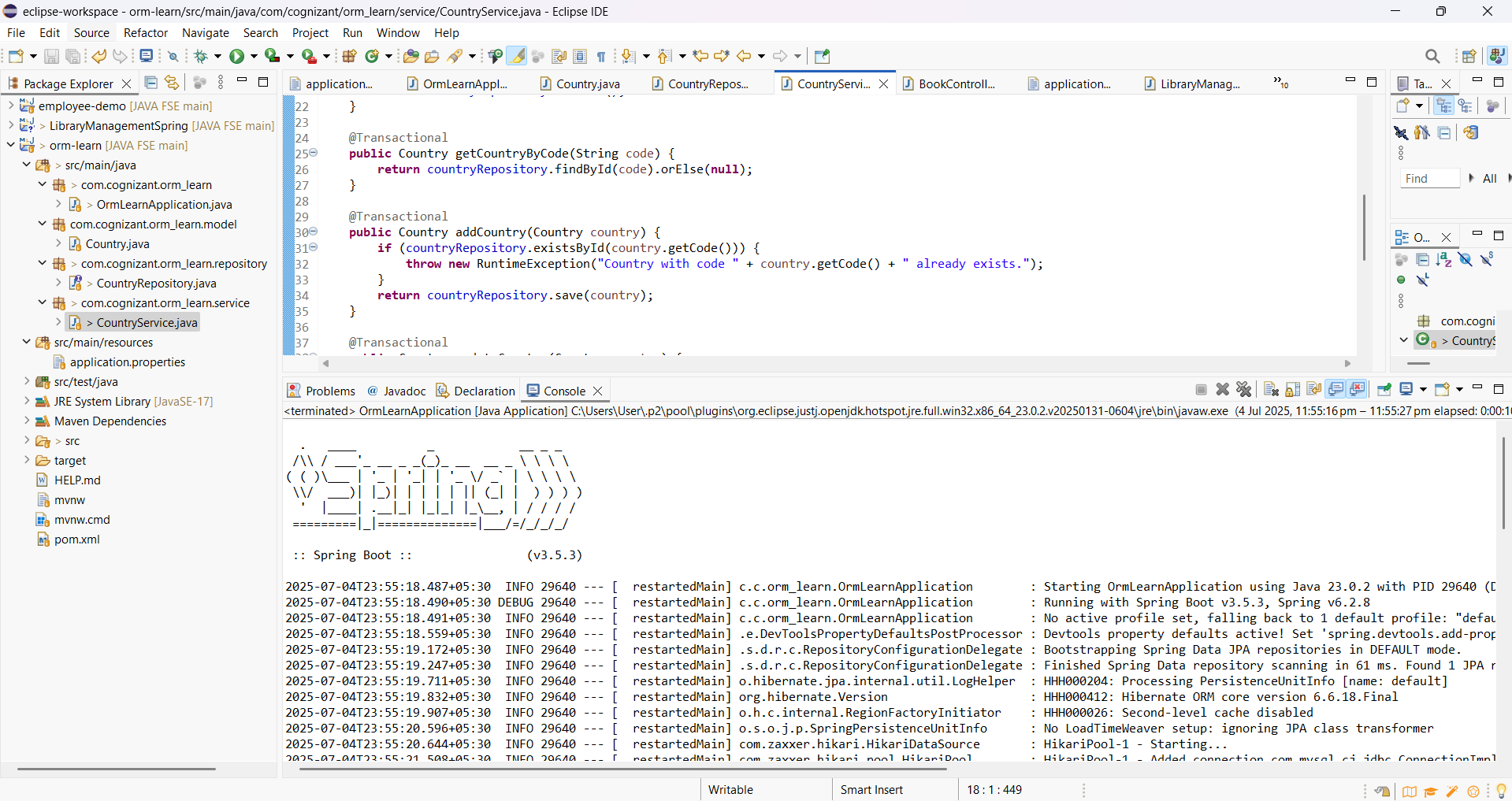
**Conclusion**

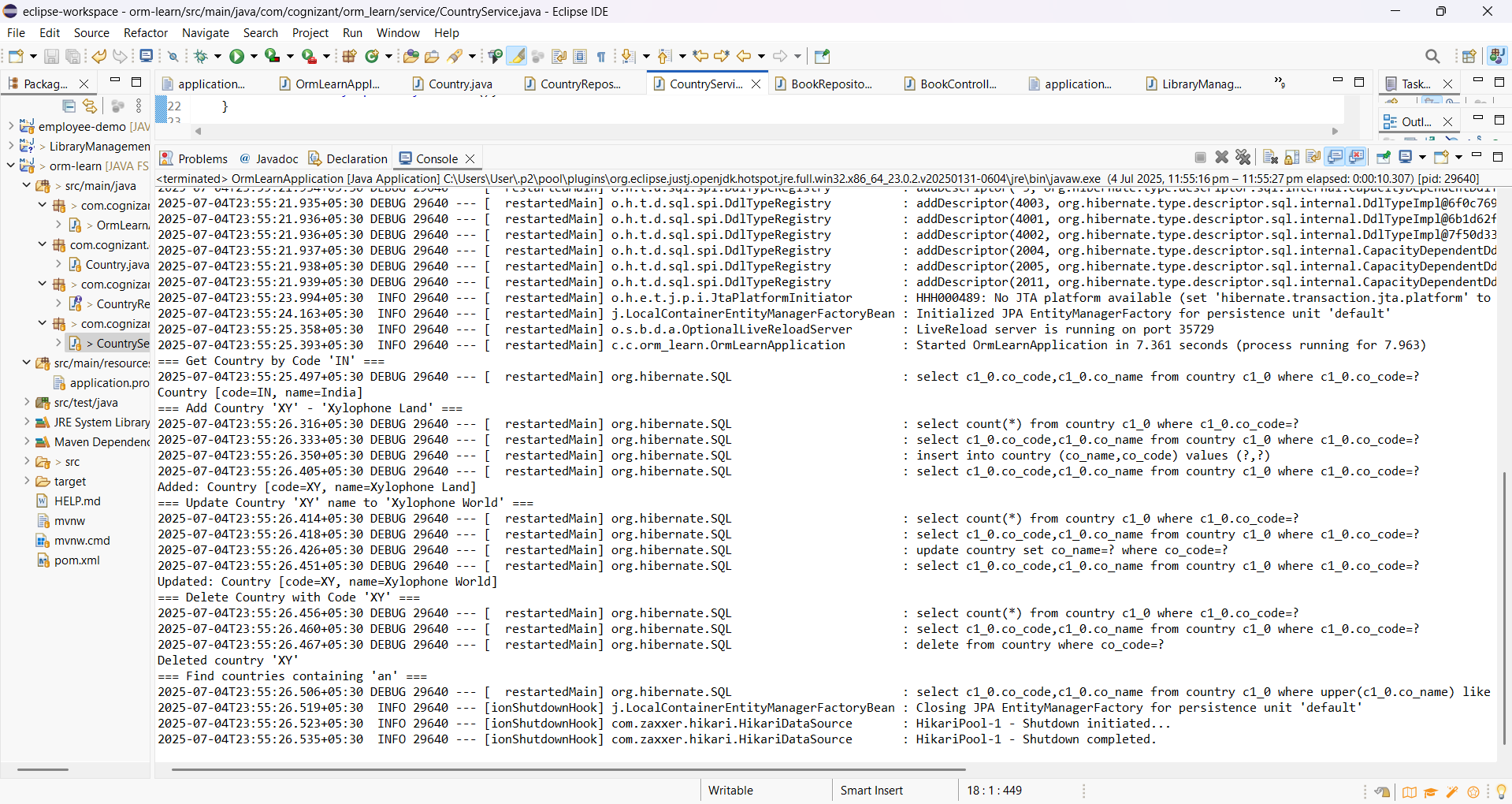
By building on my knowledge from Hands on 1, I developed a complete Country Management service with Spring Boot, Spring Data JPA, and MySQL. This includes full CRUD and search functionality, database integration, and basic in-app testing.

Future enhancements could include:

* Creating REST controllers to expose APIs
* Adding proper exception handling and validations
* Writing automated unit and integration tests
* Implementing pagination and sorting for search results

**This documentation can serve as a guide and proof of my learning and implementation for Spring Data JPA country service management.**

**Outputs:**



**2. spring-data-jpa-handson**

**What are Query Methods in Spring Data JPA?**

**Query Methods** allow to define methods in repository interfaces that automatically generate queries based on their name. The framework parses method names and constructs database queries without writing explicit JPQL or SQL.

We can do searches like:

* Find entities where some field contains text
* Sort results
* Filter by entities whose fields start with specified text
* Find between date ranges
* Find where value is greater than or less than a threshold
* Get top N records by some criteria

**What to do / How to implement**

**Prerequisites:**

* A Spring Boot project with Spring Data JPA dependency
* Entity class (e.g., **Employee**, **Product**, etc.)
* A corresponding repository interface that extends **JpaRepository<EntityClass, IDType>**

**Step 1: Define Entity**

Example:

import javax.persistence.\*;

import java.time.LocalDate;

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String firstName;

private String lastName;

private String department;

private LocalDate joiningDate;

private Double salary;

// Getters, setters, constructors

}

**Step 2: Create the Repository Interface**

Create an interface **EmployeeRepository** in the **repository** package that extends JpaRepository:

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

import java.time.LocalDate;

import java.util.List;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Search by containing text in firstName

List<Employee> findByFirstNameContaining(String text);

// Sort by lastName ascending

List<Employee> findByDepartmentOrderByLastNameAsc(String department);

// Filter with starting text on lastName

List<Employee> findByLastNameStartingWith(String prefix);

// Fetch between joiningDate

List<Employee> findByJoiningDateBetween(LocalDate startDate, LocalDate endDate);

// Greater than salary

List<Employee> findBySalaryGreaterThan(Double salary);

// Less than salary

List<Employee> findBySalaryLessThan(Double salary);

// Top 5 highest salary employees

List<Employee> findTop5ByOrderBySalaryDesc();

}

**Step 3: Using Repository Methods in Service or Controller**

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public List<Employee> searchByFirstName(String name) {

return employeeRepository.findByFirstNameContaining(name);

}

// Similar methods for other query methods

}

**Step 4: Run your application and test**

We can write unit tests or use REST controllers to call these methods and verify the results.

**Demonstrate Implementation of O/R Mapping with JPA Annotations**

**Overview**

O/R (Object-Relational) Mapping is a key concept in JPA and Spring Data JPA that maps Java object relationships to database table relationships. The most common relationships are:

* **One-to-Many / Many-to-One**
* **Many-to-Many**

These mappings define how entities relate to each other and how data is fetched and stored in relational databases.

**Core Annotations & Concepts**

**1. @ManyToOne**

* Defines many instances of an entity associated with one instance of another entity.
* Usually a foreign key exists on the many-side table.

**2. @JoinColumn**

* Specifies the foreign key column in the database on the owning side of the relationship.
* Used in conjunction with **@ManyToOne** or **@OneToOne**.

**3. @OneToMany**

* Defines one instance related to many instances of another entity.
* Typically mapped by the owning side (**mappedBy** attribute).
* It is the inverse side of a **@ManyToOne** relationship.

**4. Fetch Types**

* **FetchType.EAGER** — related entities are fetched immediately with the queried entity.
* **FetchType.LAZY** — related entities are fetched only when accessed (lazy loading).

**5. @ManyToMany**

* Defines many entities related to many others.
* Requires a join table in the database to hold relationship keys.

**6. @JoinTable**

* Specifies the join table for a many-to-many relationship.
* Defines join columns and inverse join columns (foreign keys).

**7. mappedBy**

* Used on the inverse side of bidirectional relationships.
* Refers to the field name on the owning side; avoids duplicate foreign keys or join tables.

**Example: Employee and Department Relationship**

Assuming:

* Each **Employee** belongs to one **Department** (**ManyToOne**)
* Each **Department** has many **Employees** (**OneToMany**)

**Entity: Department**

@Entity

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

// OneToMany mapped by the 'department' field in Employee entity

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

private List<Employee> employees;

// getters and setters

}

**Entity: Employee**

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String firstName;

private String lastName;

// Many employees belong to one department, with foreign key 'department\_id'

@ManyToOne(fetch = FetchType.EAGER)

@JoinColumn(name = "department\_id")

private Department department;

// getters and setters

}

**Example: Many-to-Many Relationship (Student - Course)**

Assume:

* A student can enroll in many courses.
* A course can have many students.

**Entity: Student**

@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 Set<Course> courses;

// getters and setters

}

**Entity: Course**

@Entity

public class Course {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

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

private Set<Student> students;

// getters and setters

}

**Key Points and Best Practices**

* Use **@ManyToOne + @JoinColumn** on the owning side (many-side) in a one-to-many relationship.
* Use **mappedBy** on the inverse side (one-side) with **@OneToMany** to indicate the owning side field.
* Choose **FetchType.LAZY** by default to optimize performance, especially on collection relationships.
* Use **@ManyToMany** with **@JoinTable** to define the join table for many-to-many relationships.
* Bidirectional relationships must have one owning side and one inverse side to avoid inconsistent foreign keys.

**3. spring-data-jpa-handson**

**Demonstrate Writing Hibernate Query Language (HQL) and Native Query in Spring Data JPA**

**Overview**

In Spring Data JPA, querying the database can be done in multiple ways beyond simple method name conventions:

* Using **Hibernate Query Language (HQL)** or **Java Persistence Query Language (JPQL)**
* Using **Native SQL queries** (direct database SQL)
* Using the **@Query** annotation to define custom queries

**What is HQL and JPQL?**

* **HQL (Hibernate Query Language)** is Hibernate’s own object-oriented query language.
* **JPQL (Java Persistence Query Language)** is the standardized query language defined by the JPA specification.
* Both are very similar in syntax and concepts and are used to query entities instead of tables.
* HQL queries are portable but Hibernate-specific features (e.g., HQL keywords like **fetch**) are Hibernate extensions.
* JPQL is standard across all JPA implementations.

**Key difference**:  
HQL may support extra features specific to Hibernate (e.g., **fetch** for eager fetching), whereas JPQL focuses on standard JPA features.

**Using @Query Annotation in Spring Data JPA**

* **@Query** lets you define JPQL or HQL queries directly on repository methods.
* You can specify JPQL/HQL by default or use **nativeQuery=true** to write native SQL.

**Example: Writing HQL/JPQL Query with @Query**

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// JPQL/HQL query to find employees by department name using a join

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

List<Employee> findByDepartmentName(String deptName);

}

* This query returns **Employee** entities from the entity model.
* The query is in terms of entity fields and relationships, not tables and columns.

**Using HQL fetch Keyword**

* Hibernate supports **fetch** keyword to eagerly fetch associated collections or entities.

Example:

@Query("SELECT d FROM Department d LEFT JOIN FETCH d.employees WHERE d.id = ?1")

Department findDepartmentWithEmployees(Long id);

* This fetches a Department *with* its employees initialized eagerly in a single query.

**Aggregate Functions in HQL**

* HQL supports aggregate functions such as **count()**, **sum()**, **avg()**, **max()**, and **min()**.

Example:

@Query("SELECT count(e) FROM Employee e WHERE e.department.name = ?1")

Long countEmployeesByDepartment(String deptName);

**Writing Native Queries**

* Native queries are raw SQL statements executed directly against the database.
* Useful when JPQL/HQL cannot express complex SQL or uses database-specific features.
* Use **nativeQuery=true** in **@Query**.

Example:

@Query(value = "SELECT \* FROM employee WHERE department\_id = ?1", nativeQuery = true)

List<Employee> findEmployeesByDepartmentNative(Long departmentId);

**Summary Table: HQL/JPQL vs Native Queries**

| **Feature** | **HQL / JPQL** | **Native Query** |
| --- | --- | --- |
| Language | Object-oriented (entity classes, fields) | SQL (database-specific syntax) |
| Portability | Portable across JPA implementations | Database-specific, less portable |
| Supported by @Query | Default query language | Specify **nativeQuery = true** |
| Supports Joins | Supports entity relationships | Supports any SQL join |
| Fetching | Supports **fetch** keyword (Hibernate) | Depends on SQL query |
| Aggregate Functions | Supported (**count()**, **sum()**, etc.) | Supported as per SQL |

**Hands-on 1: Introduction to HQL and JPQL**

* **HQL (Hibernate Query Language)** and **JPQL (Java Persistence Query Language)** are object-oriented query languages designed for querying data from Java objects/entities.
* Both HQL and JPQL have SQL-like syntax but operate on entity objects rather than database tables.
* **JPQL** is a subset of **HQL**:
  + All JPQL queries are valid HQL queries.
  + Not all HQL queries are valid JPQL queries.
* Supported operations in both HQL and JPQL include **SELECT**, **UPDATE**, and **DELETE**.
* HQL uniquely supports the **INSERT** statement, which JPQL does not.

**Hands-on 2 — Get All Permanent Employees Using HQL**

**✅ Objective**

Retrieve all **permanent employees** using HQL (Hibernate Query Language), and also fetch their respective **department** and **skill list** details.

**🛠️ Steps to be followed**

**1️⃣ Add method in EmployeeRepository.java**

Add the following method using @Query annotation to fetch all permanent employees along with their department and skills:

@Query(value = "SELECT e FROM Employee e left join fetch e.department d left join fetch e.skillList WHERE e.permanent = 1")

List<Employee> getAllPermanentEmployees();

**Explanation:**

* SELECT e FROM Employee e: Selects Employee objects.
* left join fetch e.department d: Joins department and fetches it.
* left join fetch e.skillList: Joins skills and fetches them.
* WHERE e.permanent = 1: Condition to get only permanent employees.

**2️⃣ Create a service method in EmployeeService.java**

Add a service method to call this repository method:

public List<Employee> getAllPermanentEmployees() {

return employeeRepository.getAllPermanentEmployees();

}

**3️⃣ Add a test method in OrmLearnApplication.java**

Create a test method to call the service and log the results:

public static void testGetAllPermanentEmployees() {

LOGGER.info("Start");

List<Employee> employees = employeeService.getAllPermanentEmployees();

LOGGER.debug("Permanent Employees:{}", employees);

employees.forEach(e -> LOGGER.debug("Skills:{}", e.getSkillList()));

LOGGER.info("End");

}

**4️⃣ Call the test method from main method**

In your main method of OrmLearnApplication.java, call the test method like this:

testGetAllPermanentEmployees();

**5️⃣ Check application logs**

After running the application, check your console logs. You should see:

* List of all permanent employees.
* Their department details.
* Their skill lists.

**⚡ Expected SQL queries in logs**

When using left join fetch, only **one optimized query** is fired. Example:

select employee0\_.em\_id as em\_id1\_2\_0\_, department1\_.dp\_id as dp\_id1\_1\_1\_, skill3\_.sk\_id as sk\_id1\_4\_2\_,

employee0\_.em\_date\_of\_birth as em\_date\_2\_2\_0\_, employee0\_.em\_dp\_id as em\_dp\_id6\_2\_0\_, employee0\_.em\_name as em\_name3\_2\_0\_,

employee0\_.em\_permanent as em\_perma4\_2\_0\_, employee0\_.em\_salary as em\_salar5\_2\_0\_, department1\_.dp\_name as dp\_name2\_1\_1\_,

skill3\_.sk\_name as sk\_name2\_4\_2\_, skilllist2\_.es\_em\_id as es\_em\_id1\_3\_0\_\_, skilllist2\_.es\_sk\_id as es\_sk\_id2\_3\_0\_\_

from employee employee0\_

left outer join department department1\_ on employee0\_.em\_dp\_id=department1\_.dp\_id

left outer join employee\_skill skilllist2\_ on employee0\_.em\_id=skilllist2\_.es\_em\_id

left outer join skill skill3\_ on skilllist2\_.es\_sk\_id=skill3\_.sk\_id

where employee0\_.em\_permanent=1

**💡 Conclusion**

After completing this hands-on:

* We are able to fetch all permanent employees.
* Their department and skills are fetched together efficiently.
* We can verify the data using logs.

**Hands-on 3 — Fetch Quiz Attempt Details Using HQL**

**✅ Objective**

To implement a feature where the admin can view details of a quiz attempted by a user, including:

* Username
* Attempt date
* Questions in the attempt
* List of options per question
* Correct answer
* Score per correct answer
* User’s selected option

**🗺️ Schema Setup**

**Steps to set up schema:**

1️⃣ Open folder spring-data-jpa-files.  
2️⃣ Open quiz.mwb file using **MySQL Workbench**.  
3️⃣ Go to File ➡ Export ➡ Forward Engineer SQL CREATE Script.  
4️⃣ Browse and select a location to save the SQL file.  
5️⃣ Check **"Generate INSERT Statements for Tables"**.  
6️⃣ Click **Next > Next > Finish** to generate.  
7️⃣ Execute the generated SQL file in ormlearn schema.  
8️⃣ Check the tables and data in your DB.

**🏗️ Entity Classes**

Create entity classes for each table:

* User
* Attempt
* Question
* Option
* AttemptQuestion
* AttemptOption

**Important points:**

* Define proper @OneToMany, @ManyToOne, @JoinColumn, and @JoinTable relationships based on schema.
* Map Attempt to User.
* Map AttemptQuestion to Attempt and Question.
* Map AttemptOption to AttemptQuestion and Option.

**🗃️ Repository Layer**

**Create AttemptRepository.java**

public interface AttemptRepository extends JpaRepository<Attempt, Integer> {

@Query("SELECT a FROM Attempt a "

+ "left join fetch a.user u "

+ "left join fetch a.attemptQuestions aq "

+ "left join fetch aq.question q "

+ "left join fetch aq.attemptOptions ao "

+ "left join fetch ao.option o "

+ "WHERE u.id = :userId AND a.id = :attemptId")

Attempt getAttempt(@Param("userId") int userId, @Param("attemptId") int attemptId);

}

**💼 Service Layer**

**Create AttemptService.java**

@Service

public class AttemptService {

@Autowired

private AttemptRepository attemptRepository;

public Attempt getAttempt(int userId, int attemptId) {

return attemptRepository.getAttempt(userId, attemptId);

}

}

**🧪 Test Method**

**Add test method in OrmLearnApplication.java**

public static void testGetAttemptDetail() {

LOGGER.info("Start");

Attempt attempt = attemptService.getAttempt(1, 1); // Pass appropriate userId and attemptId

LOGGER.debug("Username: {}", attempt.getUser().getName());

LOGGER.debug("Attempted Date: {}", attempt.getAttemptDate());

for (AttemptQuestion aq : attempt.getAttemptQuestions()) {

LOGGER.debug("Question: {}", aq.getQuestion().getDescription());

int optionNum = 1;

for (AttemptOption ao : aq.getAttemptOptions()) {

String optDesc = ao.getOption().getDescription();

Double score = ao.getOption().getScore();

Boolean isSelected = ao.getSelected();

LOGGER.debug("{} ) {} \t {} \t {}", optionNum++, optDesc, score, isSelected);

}

}

LOGGER.info("End");

}

**Call the test method**

In main method:

testGetAttemptDetail();

**💬 Expected Output (Log Format)**

Username: John

Attempted Date: 2024-06-30

Question: What is the extension of the hyper text markup language file?

1) .xhtm 0.0 false

2) .ht 0.0 false

3) .html 1.0 true

4) .htmx 0.0 false

Question: What is the maximum level of heading tag can be used in a HTML page?

1) 5 0.0 false

2) 3 0.0 true

3) 4 0.0 false

4) 6 1.0 false

...

**⚡ HQL Explanation**

* The HQL joins all required tables:
  + User
  + Attempt
  + AttemptQuestion
  + Question
  + AttemptOption
  + Option
* Condition applied: WHERE u.id = :userId AND a.id = :attemptId
* fetch used in each join to populate nested relationships and avoid N+1 problem.

**💡 Conclusion**

After completing this hands-on:

* Admin can fetch and view detailed quiz attempt data.
* The solution is optimized using HQL with fetch joins.
* Display format matches expected output.

**Hands-on 4 — Get Average Salary Using HQL**

**✅ Objective**

Compute the **average salary** of a department using HQL.

**🛠️ Steps to be followed**

**1️⃣ Add method in EmployeeRepository.java**

First, add a basic average salary query:

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

double getAverageSalary();

But this does **not** filter by department. To calculate based on department, modify it as below:

@Query(value = "SELECT AVG(e.salary) FROM Employee e WHERE e.department.id = :id")

double getAverageSalary(@Param("id") int id);

**2️⃣ Add method in EmployeeService.java**

public double getAverageSalary(int id) {

return employeeRepository.getAverageSalary(id);

}

**3️⃣ Add test method in OrmLearnApplication.java**

public static void testGetAverageSalary() {

LOGGER.info("Start");

double avgSalary = employeeService.getAverageSalary(1); // Pass appropriate department id

LOGGER.debug("Average Salary: {}", avgSalary);

LOGGER.info("End");

}

**4️⃣ Call the test method**

In main method:

testGetAverageSalary();

**💬 Expected Output (Logs)**

Average Salary: 35000.0

*(Actual value will depend on your data.)*

**Hands-on 5 — Get All Employees Using Native Query**

**✅ Objective**

Retrieve all employees using **Native SQL Query**.

**🛠️ Steps to be followed**

**1️⃣ Add method in EmployeeRepository.java**

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

List<Employee> getAllEmployeesNative();

**2️⃣ Add method in EmployeeService.java**

public List<Employee> getAllEmployeesNative() {

return employeeRepository.getAllEmployeesNative();

}

**3️⃣ Add test method in OrmLearnApplication.java**

public static void testGetAllEmployeesNative() {

LOGGER.info("Start");

List<Employee> employees = employeeService.getAllEmployeesNative();

LOGGER.debug("Employees: {}", employees);

LOGGER.info("End");

}

**4️⃣ Call the test method**

In main method:

testGetAllEmployeesNative();

**💬 Expected Output (Logs)**

Employees: [Employee(id=1, name=John, ...), Employee(id=2, name=Mike, ...)]

*(Exact output depends on your database records.)*

**✅ Conclusion**

After completing these hands-ons:

* You can compute department-wise average salaries using HQL.
* You can fetch all employees using a native SQL query.
* You practiced both HQL and native SQL approaches in Spring Data JPA.

**Hands-on 6 — Criteria Query**

**✅ Objective**

Understand how to implement **dynamic filtering** of products (like in an online retail site such as Amazon) using **Criteria Query** in Hibernate / JPA.

**🛍️ Scenario Explanation**

* User visits an online retail website (e.g., Amazon).
* Searches for **"laptop"**.
* On the left side, user sees filter categories such as:
  + Customer Review
  + Hard Disk Size
  + RAM Size
  + CPU Speed
  + Operating System
  + Weight
  + CPU Type
* User can select one or more filters and perform the search.

**❓ What will the WHERE clause look like in HQL?**

* The **WHERE clause** depends on which filters the user selects.
* Example if user chooses:
  + RAM Size: 16 GB
  + CPU: Intel i7
  + Customer Review: 4 stars & above

Then WHERE clause could be:

WHERE p.ramSize = '16 GB' AND p.cpu = 'Intel i7' AND p.rating >= 4

But since filters are optional and vary, it is **not possible to hardcode a single WHERE clause**.

**🛠️ Why Criteria Query?**

* When user filters change dynamically, Criteria Query helps to **programmatically build** the query.
* We can add conditions only if a filter is selected.
* Easier to maintain and avoids creating many different HQL strings manually.

**💻 Example Implementation Steps**

**1️⃣ Get CriteriaBuilder from EntityManager**

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

**2️⃣ Create CriteriaQuery**

CriteriaQuery<Product> cq = cb.createQuery(Product.class);

Root<Product> product = cq.from(Product.class);

**3️⃣ Create list of predicates (conditions)**

List<Predicate> predicates = new ArrayList<>();

if (ramSize != null) {

predicates.add(cb.equal(product.get("ramSize"), ramSize));

}

if (cpu != null) {

predicates.add(cb.equal(product.get("cpu"), cpu));

}

if (rating != null) {

predicates.add(cb.greaterThanOrEqualTo(product.get("rating"), rating));

}

// Add more conditions similarly for other filters...

**4️⃣ Combine predicates to the query**

cq.select(product).where(cb.and(predicates.toArray(new Predicate[0])));

**5️⃣ Execute query**

List<Product> results = entityManager.createQuery(cq).getResultList();

**💬 Example Criteria Query code snippet (summary)**

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<Product> cq = cb.createQuery(Product.class);

Root<Product> product = cq.from(Product.class);

List<Predicate> predicates = new ArrayList<>();

if (ramSize != null) {

predicates.add(cb.equal(product.get("ramSize"), ramSize));

}

if (cpu != null) {

predicates.add(cb.equal(product.get("cpu"), cpu));

}

if (rating != null) {

predicates.add(cb.greaterThanOrEqualTo(product.get("rating"), rating));

}

cq.select(product).where(cb.and(predicates.toArray(new Predicate[0])));

List<Product> products = entityManager.createQuery(cq).getResultList();

**🟢 Advantages of Criteria Query**

✅ Avoids writing complex or dynamic string-based HQL.  
✅ Flexible and easier to add/remove conditions at runtime.  
✅ Helps maintain cleaner and more maintainable code.

**💡 Conclusion**

* In real-world dynamic search scenarios (like Amazon filters), Criteria Query is the ideal solution.
* We build **WHERE clauses dynamically**, depending on user-selected filters.
* Makes applications more user-friendly and efficient.