# Spring Data JPA with Spring Boot, Hibernate

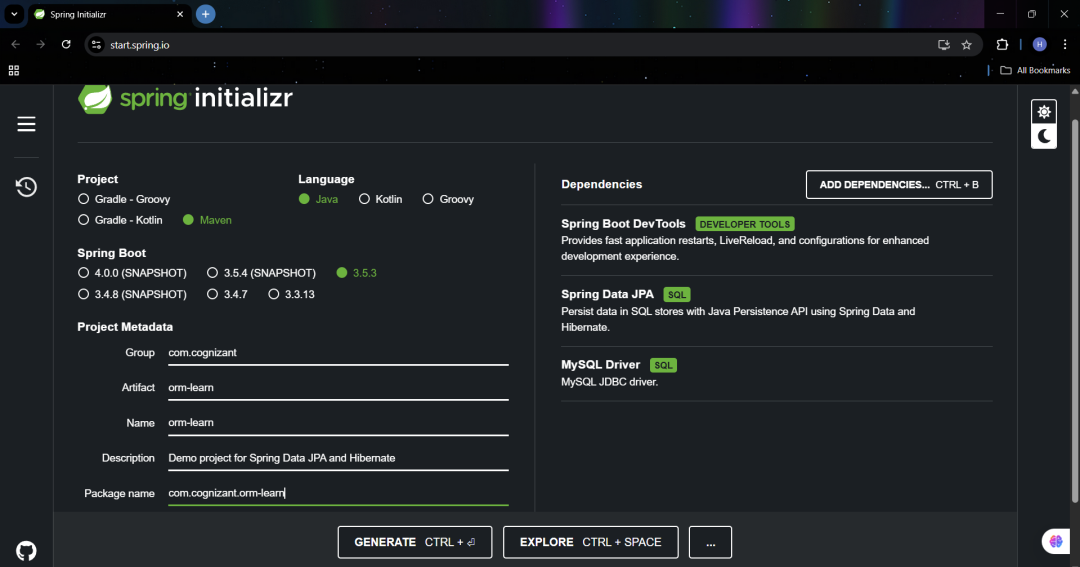
# spring-data-jpa-handson

**Mandatory hands-on**

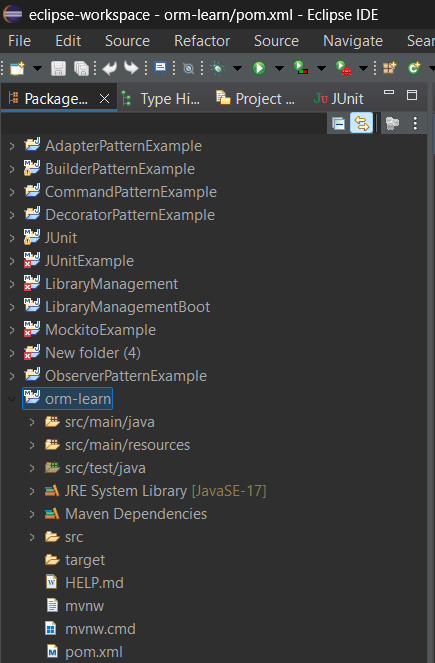
**Hands on 1**

**Spring Data JPA - Quick Example**   
  
**Software Pre-requisites**

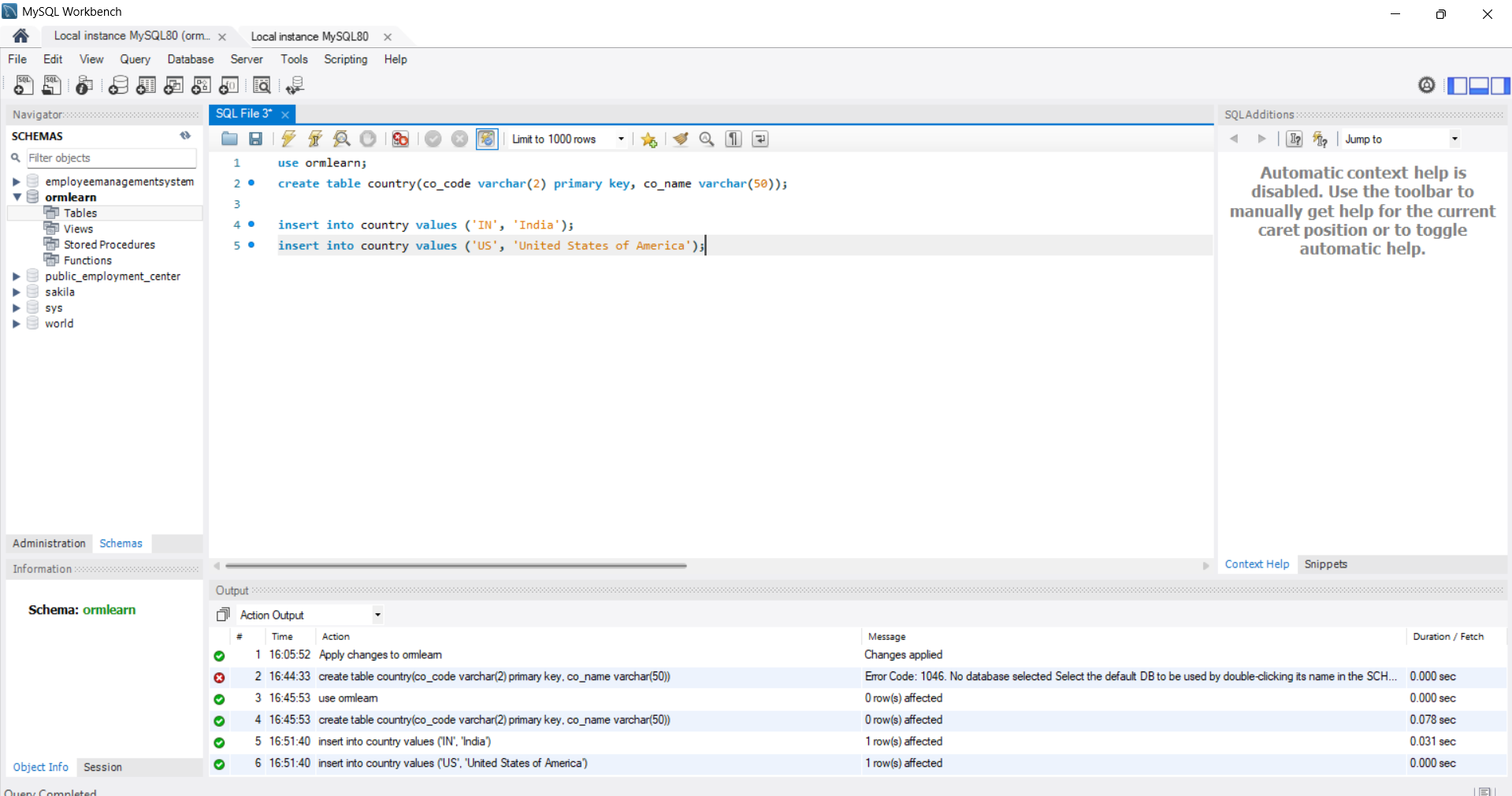
* MySQL Server 8.0
* MySQL Workbench 8
* Eclipse IDE for Enterprise Java Developers 2019-03 R
* Maven 3.6.2
* **Create a Eclipse Project using Spring Initializr**



* **Import the project in Eclipse** "File > Import > Maven > Existing Maven Projects > Click Browse and select extracted folder > Finish"



* **Create a new schema** "ormlearn" in MySQL database. Execute the following commands to open MySQL client and create schema.



* In orm-learn Eclipse project, open src/main/resources/application.properties and include the below database and log configuration.

**Application.properties**

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

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

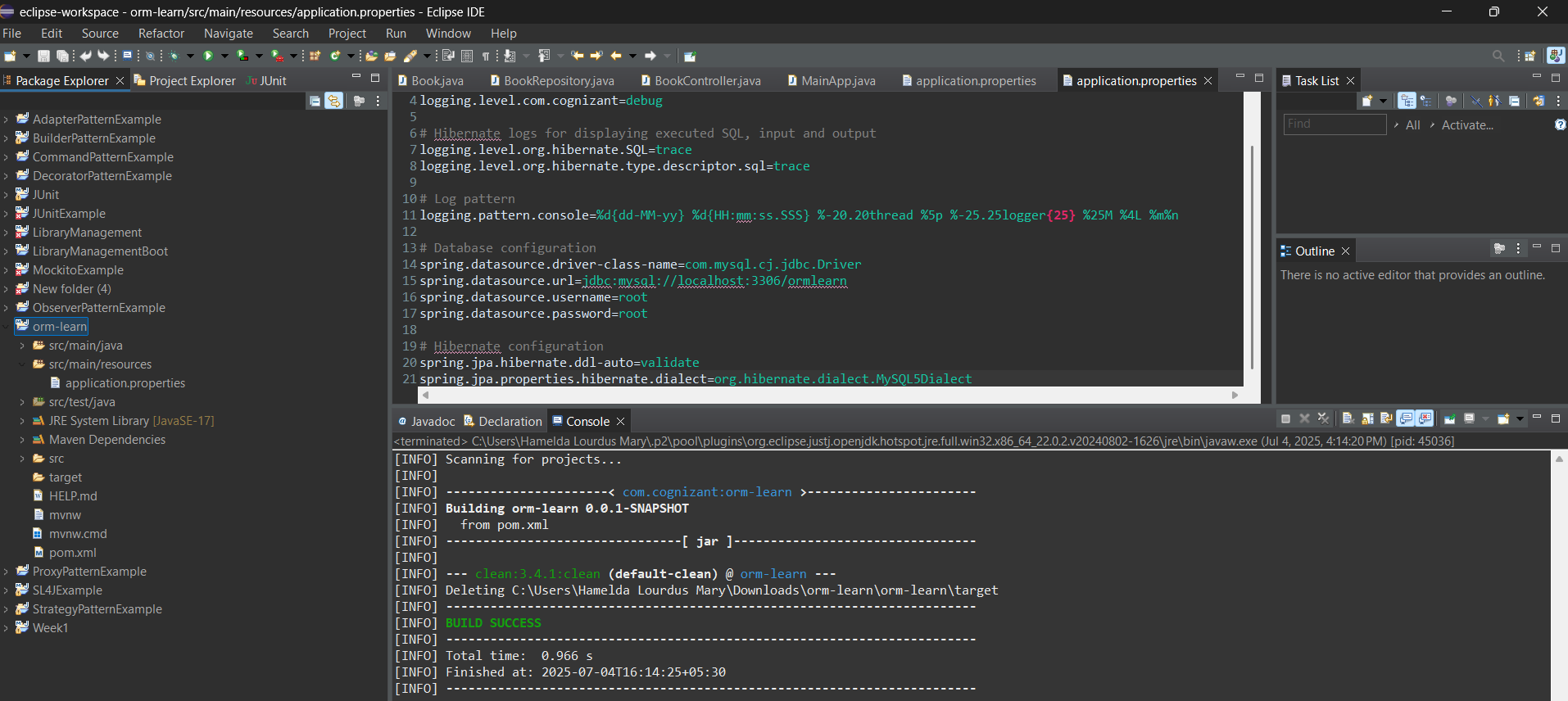
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto=update

spring.h2.console.enabled=true

server.port=8081

* **Build the project using** ‘mvn clean package’



* **src/main/java - Folder with application code**

Include logs for verifying if main() method is called.

**OrmLearnApplication.java**

package com.cognizant.orm\_learn;

import java.util.List;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

@SpringBootApplication

public class OrmLearnApplication {

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

public static void main(String[] args) {

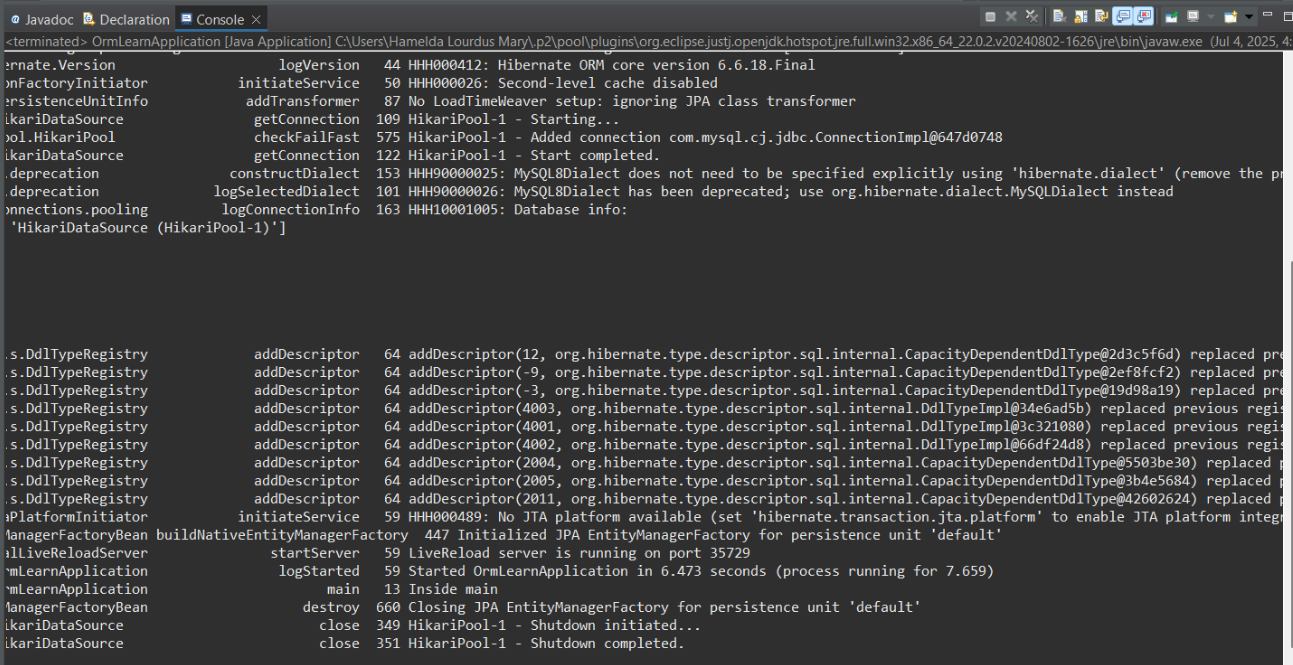
SpringApplication.run(OrmLearnApplication.class, args);

***LOGGER.info("Inside main");***  }

}

* **Execute :**

Execute the OrmLearnApplication and check in log if main method is called.



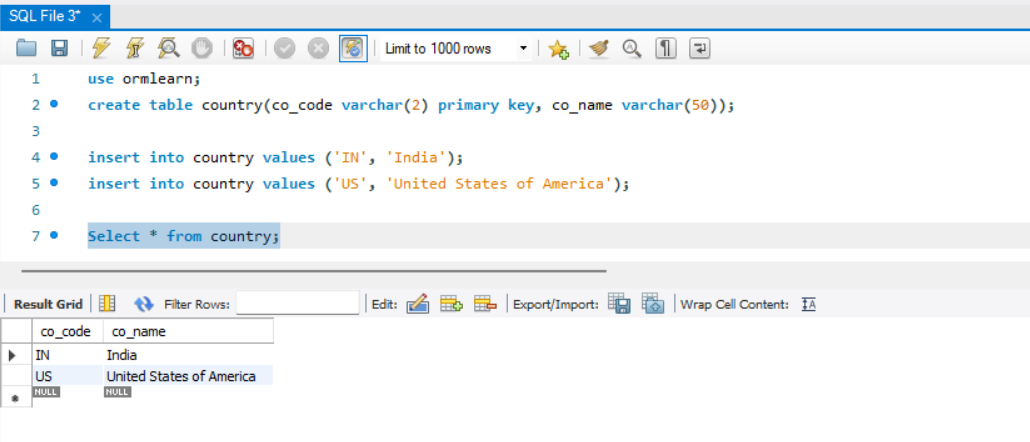
* **Country table creation**

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

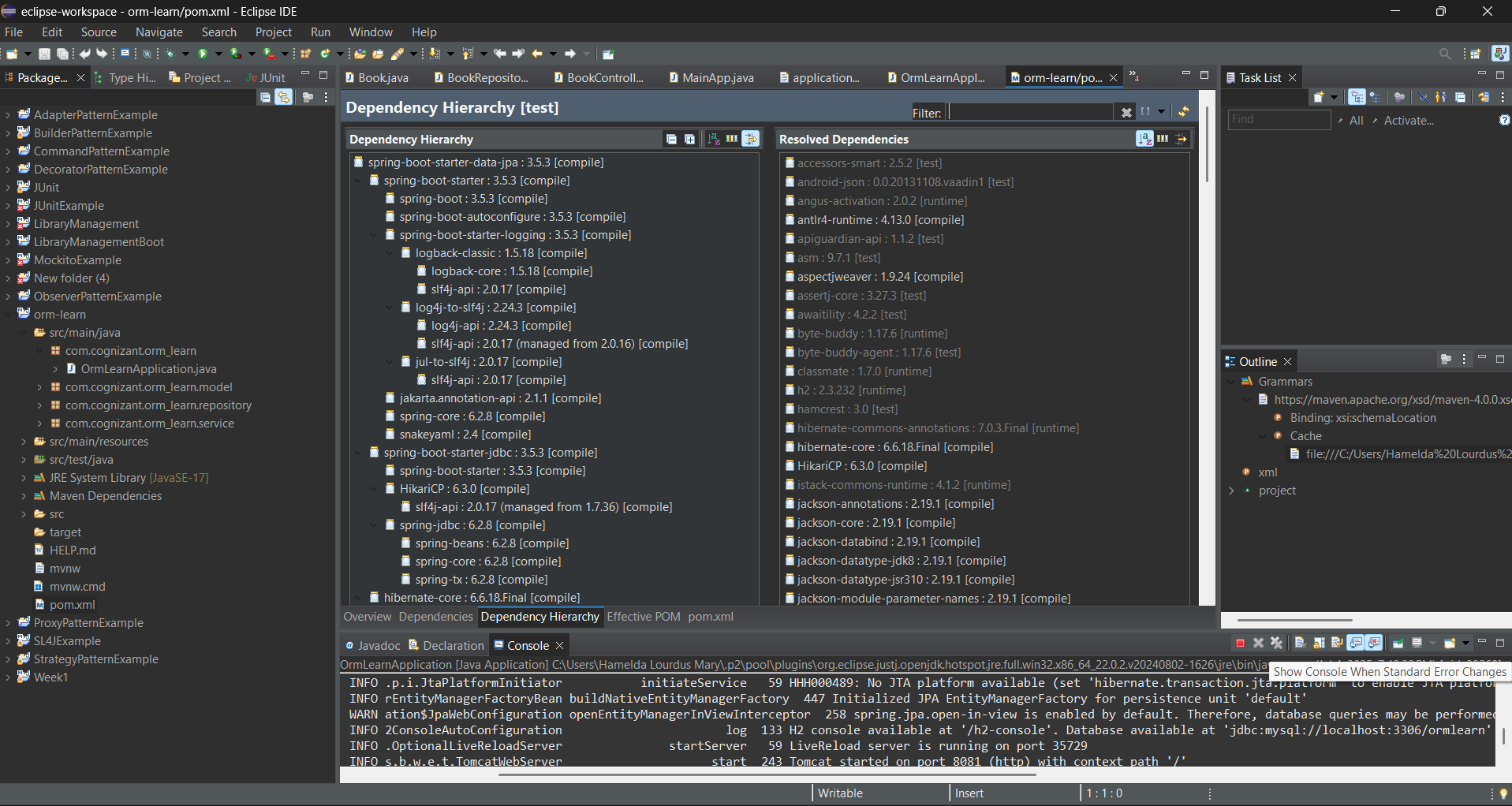
* **Insert couple of records into the table**

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

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



* **Pom.xml**
  1. Walkthrough all the configuration defined in XML file
  2. Open 'Dependency Hierarchy' and show the dependency tree.



* **Persistence Class - com.cognizant.orm-learn.model.Country**

**Country.java**

package com.cognizant.orm\_learn.model;

import jakarta.persistence.\*;

@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; }

@Override

public String toString() {

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

}

}

* **Repository Class - com.cognizant.orm-learn.CountryRepository**

**CountryRepository.java**

package com.cognizant.orm\_learn.repository;

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> {}

* **Service Class - com.cognizant.orm-learn.service.CountryService**

**CountryService.java**

package com.cognizant.orm\_learn.service;

import java.util.List;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

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();

}

}

* **OrmLearnApplication.java**

package com.cognizant.orm\_learn;

import java.util.List;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

@SpringBootApplication

public class OrmLearnApplication {

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

private static CountryService countryService;

public static void main(String[] args) {

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

LOGGER.info("Inside main");

countryService = context.getBean(CountryService.class);

testGetAllCountries();

}

private static void testGetAllCountries() {

LOGGER.info("Start");

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

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

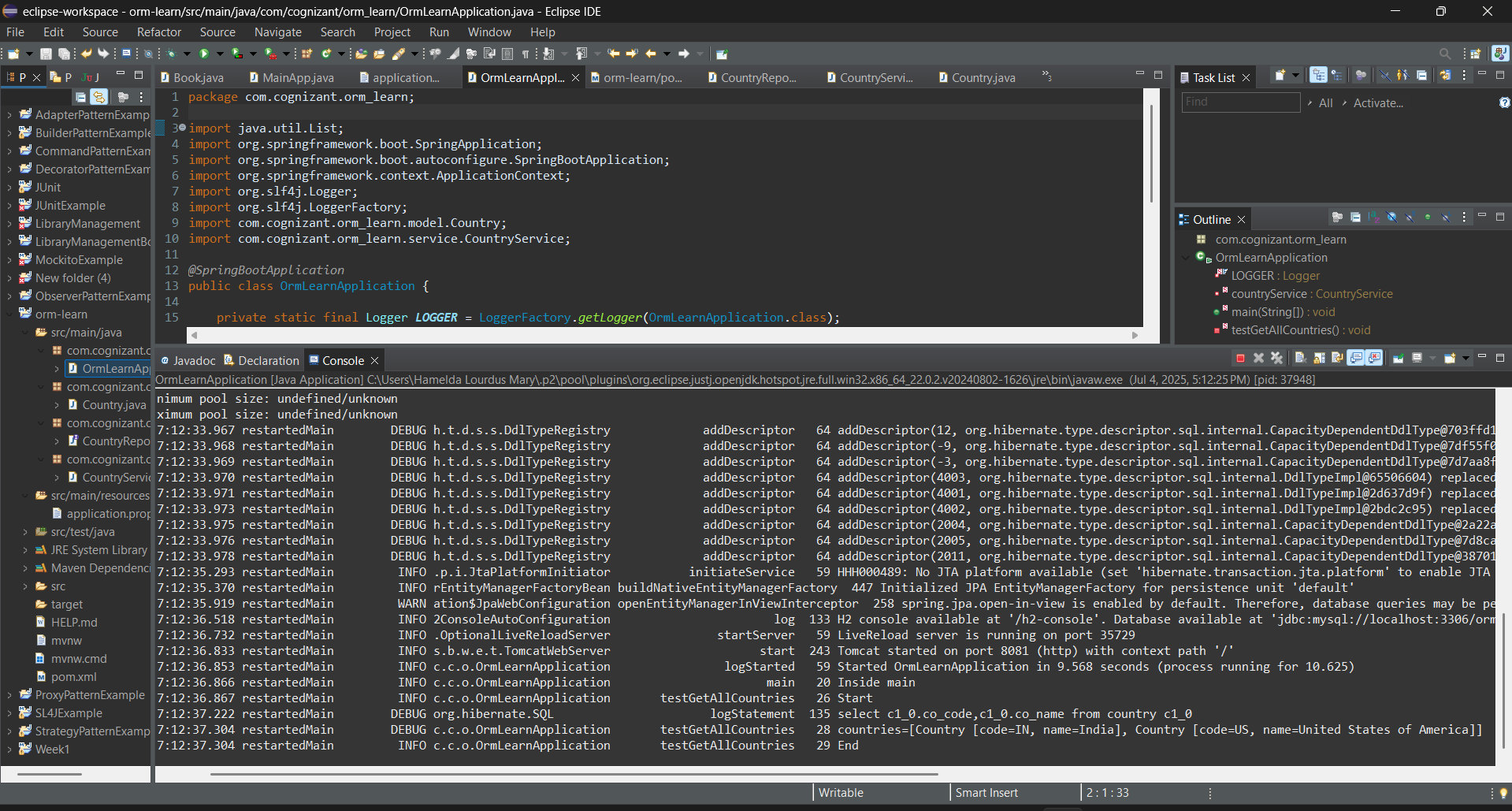
LOGGER.info("End");

}

}

* **Output:**

Data from ormlearn database is retrieved.



**Hands on 4**

**Difference between JPA, Hibernate and Spring Data JPA** 

**Introduction:**

In enterprise Java development, managing database operations efficiently is crucial. Technologies like **Java Persistence API (JPA)**, **Hibernate**, and **Spring Data JPA** play a vital role in implementing object-relational mapping (ORM) and simplifying data persistence.

**Java Persistence API (JPA):**

* The **Java Persistence API (JPA)** is a **Java specification (JSR 338)** for accessing, persisting, and managing data between Java objects and relational databases. It defines a set of interfaces and annotations such as @Entity, @Table, @Id, and @Column that guide how ORM (Object Relational Mapping) should work in Java applications.
* **JPA is only a specification and does not provide implementation**.

**Hibernate:**

**Hibernate** is a **Java-based ORM framework** that provides a reference implementation of the JPA specification. It automates the mapping of Java classes to database tables and simplifies CRUD operations. Hibernate supports advanced features like:

* Lazy loading
* Caching
* Dirty checking
* HQL (Hibernate Query Language)

In essence, Hibernate **implements JPA** and also extends it with additional capabilities beyond the standard specification

**Spring Data JPA**

**Spring Data JPA** is a part of the **Spring Data project** and offers a **higher level of abstraction over JPA**. It does **not implement JPA**, but instead provides an infrastructure to **automatically generate repository implementations**, thereby reducing boilerplate code significantly.

* Spring Data JPA works **on top of a JPA provider**, such as Hibernate or EclipseLink, and integrates smoothly with the **Spring Framework**.
* Manages transactions
* Spring Data JPA = Abstraction Layer + Auto-Generated Repositories + Spring Integration

**Difference Between them:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **JPA** | **Hibernate** | **Spring Data JPA** |
| **Type** | Specification (API) | Framework (Implementation) | Abstraction layer (Spring Module) |
| **Implementation** | No | Yes | No (uses JPA provider) |
| **Boilerplate Code** | High | Moderate | Low |
| **Transaction Management** | No | Manual | Automatic via Spring |
| **Default Query Language** | JPQL | HQL | JPQL (auto-generated) |
| **CRUD Implementation** | Manual | Manual | Auto-generated |
| **Integration with Spring** | Partial | Partial | Full |

**Code Comparison:**

### Hibernate Example (Manual ORM Using Session API)

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;

}

### Explanation:

This approach requires manually handling the Session, Transaction, exception handling, and session closing. It’s more verbose and error-prone, especially in large applications.

### Spring Data JPA Example (Declarative and Simplified)

**Repository Interface**

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {}

**Service Class**

@Servicepublic class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

employeeRepository.save(employee);

}

}

### Explanation:

With Spring Data JPA:

* The repository interface automatically provides basic CRUD operations.
* No need to manually manage sessions or transactions.
* Code is cleaner, more maintainable, and follows the **convention over configuration** principle.