**WEEK 3**

**Spring Core and Maven**

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

**Scenario:**

Your company is developing a web application for managing a library. You need to use the Spring Framework to handle the backend operations.

**CODE**

**pom.xml:**

*<?*xml version="1.0" encoding="UTF-8"*?>*<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.library</groupId>  
 <artifactId>LibraryManagement</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <properties>  
 <maven.compiler.source>17</maven.compiler.source>  
 <maven.compiler.target>17</maven.compiler.target>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 </properties>  
 <dependencies>  
 *<!-- Spring Context dependency -->* <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.3.34</version>  
 </dependency>  
 </dependencies>  
  
</project>

**applicationContext.xml:**

*<?*xml version="1.0" encoding="UTF-8"*?>*<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="  
 http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd"><bean id="bookRepository" class="com.library.repository.BookRepository"/>  
<bean id="bookService" class="com.library.service.BookService">  
 <property name="bookRepository" ref="bookRepository"/>  
 </bean>  
  
</beans>

**BookRepository.java:**

package com.library.repository;  
public class BookRepository {  
 public void saveBook(String bookName) {  
 System.*out*.println("Book '" + bookName + "' saved to the repository.");  
 }  
}

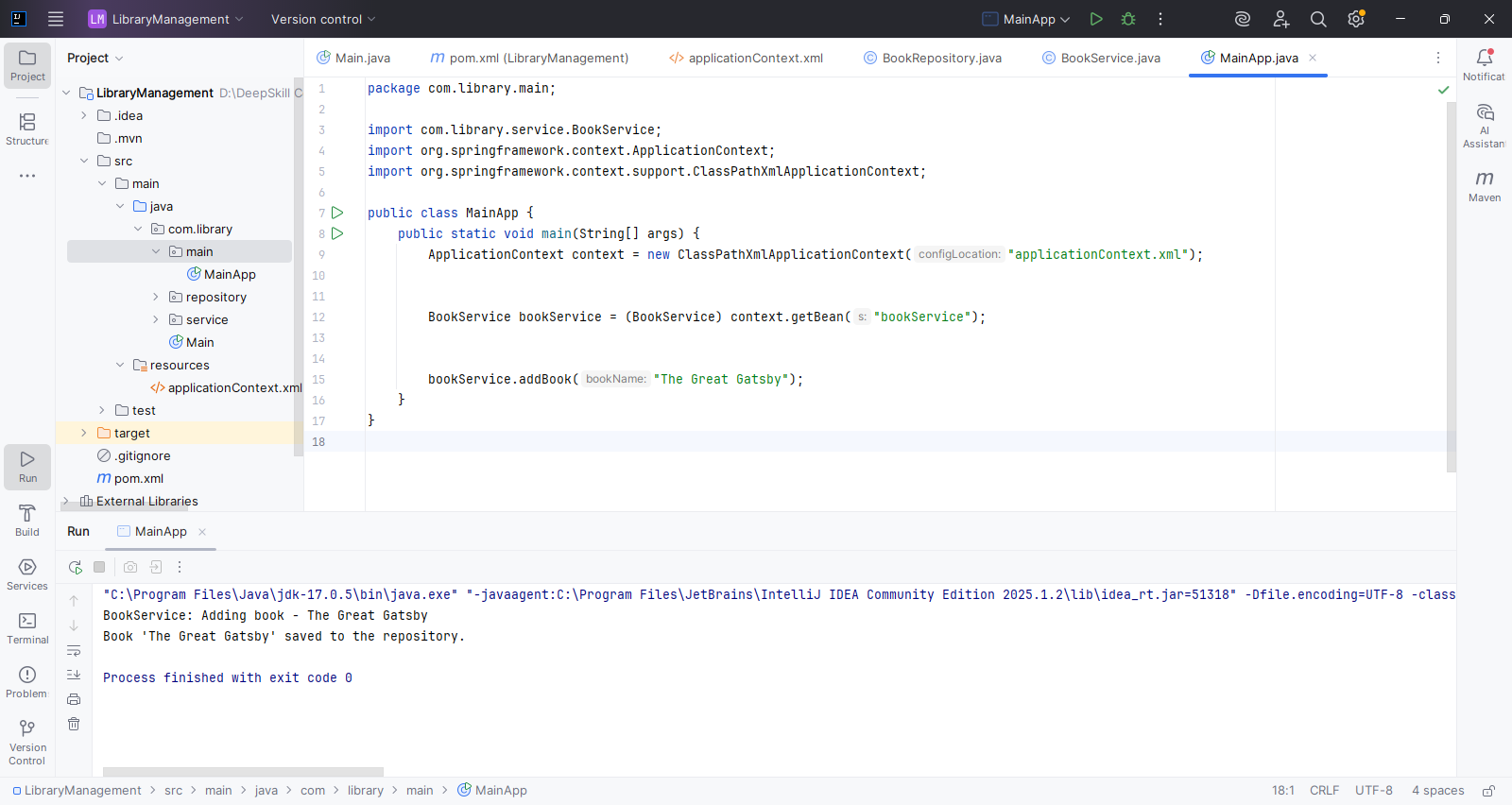
**BookService.java:**

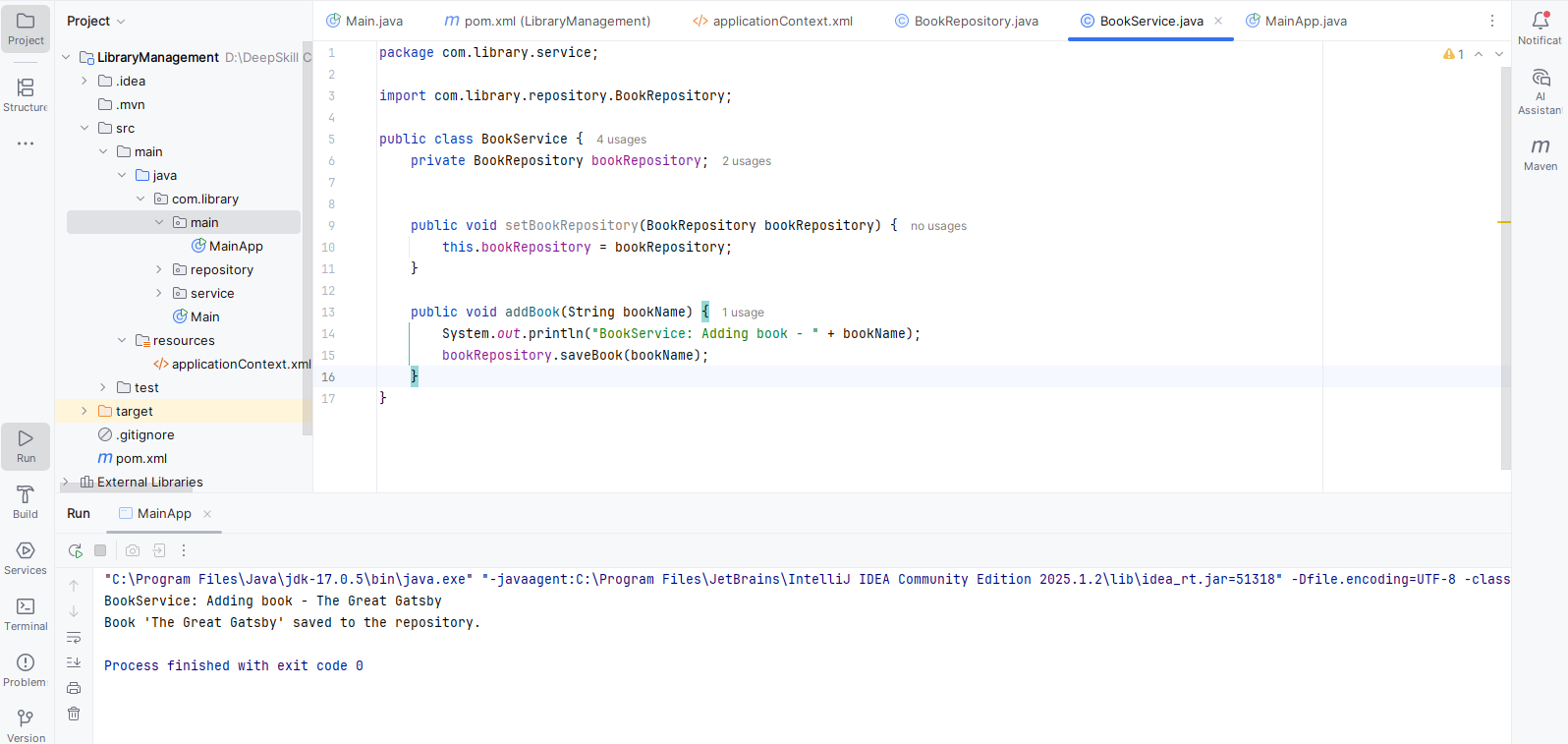
package com.library.service;  
  
import com.library.repository.BookRepository;  
  
public class BookService {  
 private BookRepository bookRepository;  
  
  
 public void setBookRepository(BookRepository bookRepository) {  
 this.bookRepository = bookRepository;  
 }  
  
 public void addBook(String bookName) {  
 System.*out*.println("BookService: Adding book - " + bookName);  
 bookRepository.saveBook(bookName);  
 }  
}

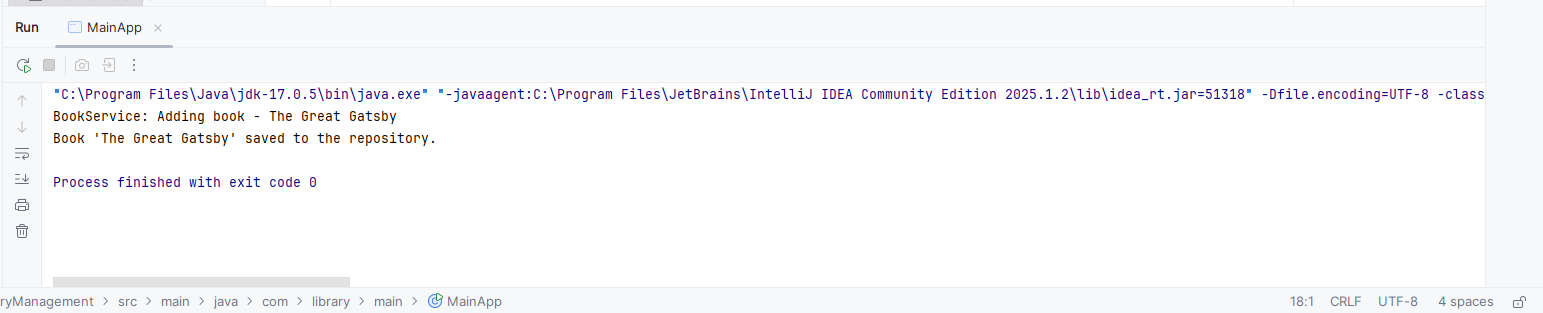
**MainApp.java:**

package com.library.main;  
  
import com.library.service.BookService;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class MainApp {  
 public static void main(String[] args) {  
 ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
 BookService bookService = (BookService) context.getBean("bookService");  
  
  
 bookService.addBook("The Great Gatsby");  
 }  
}

**SCREENSHOTS**







**Conclusion for Exercise 1:**

In this exercise, we successfully configured a basic Spring application using XML-based configuration. We followed a step-by-step approach to:

* Set up a Maven-based Java project.
* Add Spring Core dependencies to manage beans and application context.
* Define a BookService and BookRepository as POJO classes.
* Configure these classes as Spring beans in an applicationContext.xml file.
* Load and test the Spring container using a Main class.

This setup demonstrates **Spring’s Inversion of Control (IoC)** and **Dependency Injection (DI)** features. By managing objects through the Spring container instead of manually instantiating them, we achieve loose coupling and better maintainability — a foundational principle for building scalable enterprise applications.

**Exercise 2: Implementing Dependency**

***applicationContext.xml***

*<?*xml version="1.0" encoding="UTF-8"*?>*<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd">  
 <bean id="bookRepository" class="com.library.BookRepository" />  
 <bean id="bookService" class="com.library.BookService">  
 <property name="bookRepository" ref="bookRepository" />  
 </bean>  
</beans>

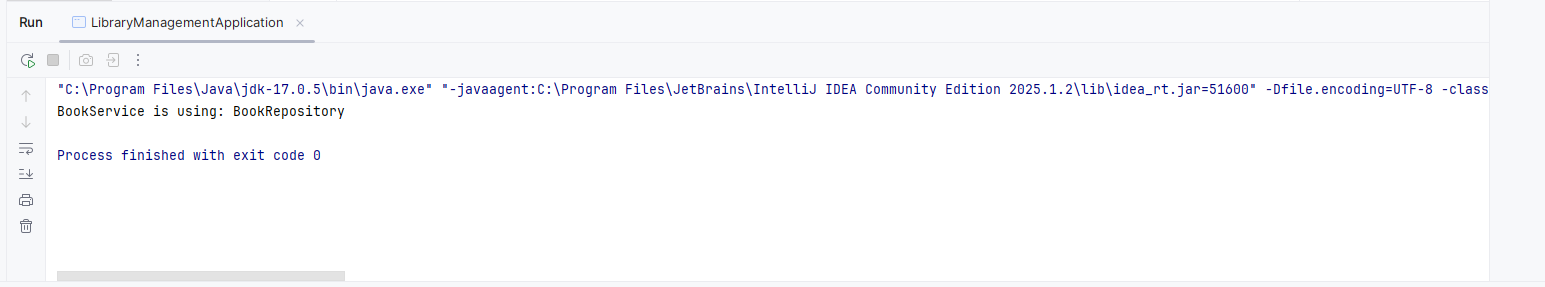
**BookService.java**

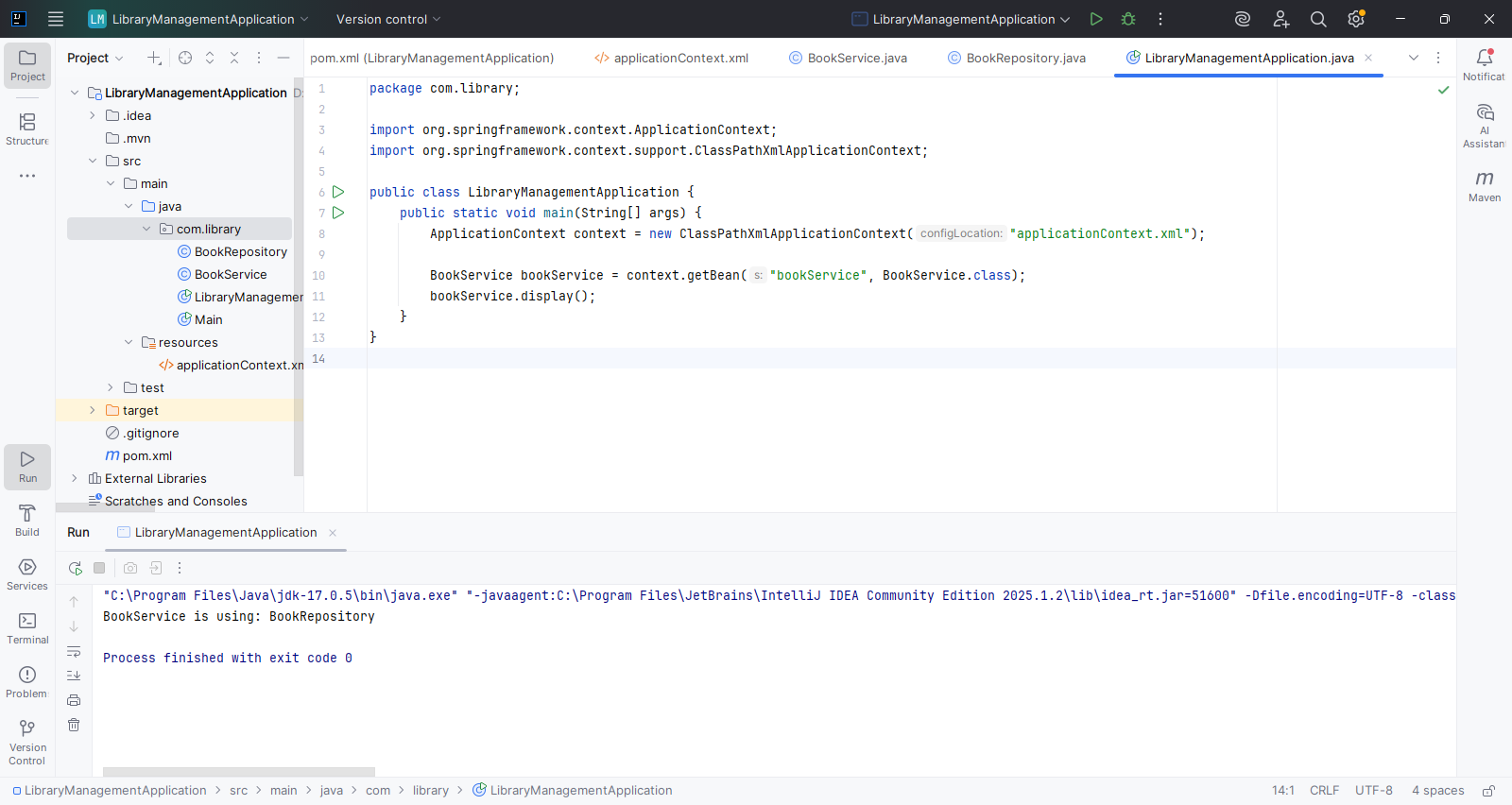
package com.library;  
public class BookService {  
  
 private BookRepository bookRepository;  
   
 public void setBookRepository(BookRepository bookRepository) {  
 this.bookRepository = bookRepository;  
 }  
 public void display() {  
 System.*out*.println("BookService is using: " + bookRepository.getClass().getSimpleName());  
 }  
}

**LibraryManagementApplication.java**

package com.library;  
  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class LibraryManagementApplication {  
 public static void main(String[] args) {  
 ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
  
 BookService bookService = context.getBean("bookService", BookService.class);  
 bookService.display();  
 }  
}

**SCREENSHOTS**





**Conclusion for exercise 2:**

This exercise demonstrates how to:

* Configure and inject dependencies in Spring using XML-based configuration.
* Use setter-based Dependency Injection.
* Separate concerns between repository and service layers, following best practices.

Spring IoC container successfully injected BookRepository into BookService, confirming the working of DI.

Exercise 4: Creating and Configuring a Maven Project

**Objective**

To create and configure a Maven project named LibraryManage and add necessary dependencies and plugins to support Spring development using Java 1.8.

**Description**

In this exercise, a new Maven project is created to serve as the base for a Spring-powered Library Management application. The objective is to properly set up the Maven build system by:

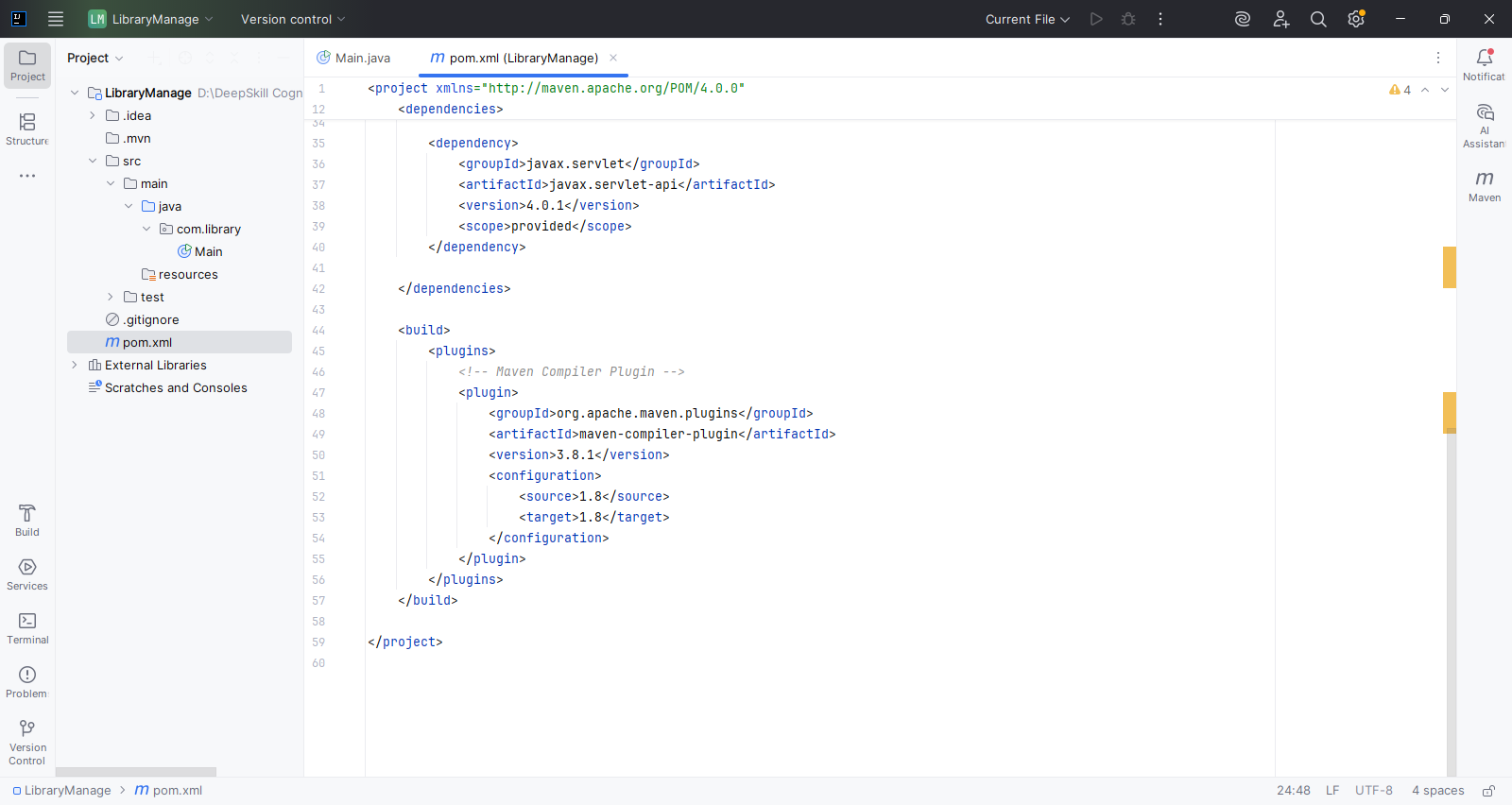
* Defining groupId, artifactId, and version for project identification.
* Adding dependencies for Spring modules (spring-context, spring-aop, and spring-webmvc) required for the application's core functionalities and MVC architecture.
* Including the Java Servlet API to support web components.
* Configuring the Maven Compiler Plugin to compile the project with Java version 1.8.

**CODE**

**pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.library</groupId>  
 <artifactId>LibraryManage</artifactId> *<!-- You said project name is LibraryManage -->* <version>1.0-SNAPSHOT</version>  
  
 <dependencies>  
  
 *<!-- Spring Context -->* <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.3.32</version>  
 </dependency>  
  
  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-aop</artifactId>  
 <version>5.3.32</version>  
 </dependency>  
  
  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>5.3.32</version>  
 </dependency>  
  
 <dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>javax.servlet-api</artifactId>  
 <version>4.0.1</version>  
 <scope>provided</scope>  
 </dependency>  
  
 </dependencies>  
  
 <build>  
 <plugins>  
 *<!-- Maven Compiler Plugin -->* <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>3.8.1</version>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 </configuration>  
 </plugin>  
 </plugins>  
 </build>  
  
</project>

**SCREENSHOT**



**Conclusion for exercise 4:**

This exercise successfully sets up the foundational structure for a Spring-based application using Maven. The pom.xml file was correctly configured with required Spring dependencies and the Maven Compiler Plugin targeting Java 1.8.

This configuration ensures that:

* Spring's core and MVC modules are available for development.
* The project compiles consistently with Java 1.8.
* The build is clean, reusable, and ready for further development such as adding beans, services, and controllers.

**Spring Data JPA with Spring Boot, Hibernate**

**Spring Data JPA - Quick Example**

This exercise demonstrates how to set up a Spring Boot application using Spring Data JPA and MySQL. The application performs basic data retrieval using repository and service layers.

**STEPS INVOLVED**

| **Step** | **Description** |
| --- | --- |
| Step 1 | Project setup using Spring Initializr |
| Step 2 | Created MySQL schema ormlearn |
| Step 3 | Configured application.properties with database details |
| Step 4 | Built the project using Maven |
| Step 5 | Added logging to verify startup |
| Step 6 | Created table country with sample data |
| Step 7 | Created entity class Country.java |
| Step 8 | Created repository interface CountryRepository |
| Step 9 | Created service class CountryService |
| Step 10 | Called the service from main() and tested |
| Step 11 | Verified the output in the console |

**CODE**

**application.properties**

# Logging - helps you see what's happening

logging.level.org.springframework=info

logging.level.com.cognizant=debug

logging.level.org.hibernate.SQL=trace

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

# Format of log output

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

# Database connection

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=Maha2003@

# Hibernate

spring.jpa.hibernate.ddl-auto=validate

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

**OrmLearnApplication.java**

package com.cognizant.orm\_learn;

import java.util.List;

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 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() - Spring Boot application started successfully!");

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

*testGetAllCountries*();

}

private static void testGetAllCountries() {

***LOGGER***.info(" Retrieving all countries...");

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

for (Country c : countries) {

***LOGGER***.info(" {}", c);

}

***LOGGER***.info(" Country fetch complete.");

}

}

**Country.java**

package com.cognizant.orm\_learn;

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;

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 + "]";

}

}

**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.Country;

*@Repository*

public interface CountryRepository extends JpaRepository<Country, String> {

}

**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.Country;

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

*@Service*

public class CountryService {

*@Autowired*

private CountryRepository countryRepository;

*@Transactional*

public List<Country> getAllCountries() {

return countryRepository.findAll();

}

}

**MySQL :**

CREATE TABLE country (

co\_code VARCHAR(2) PRIMARY KEY,

co\_name VARCHAR(50)

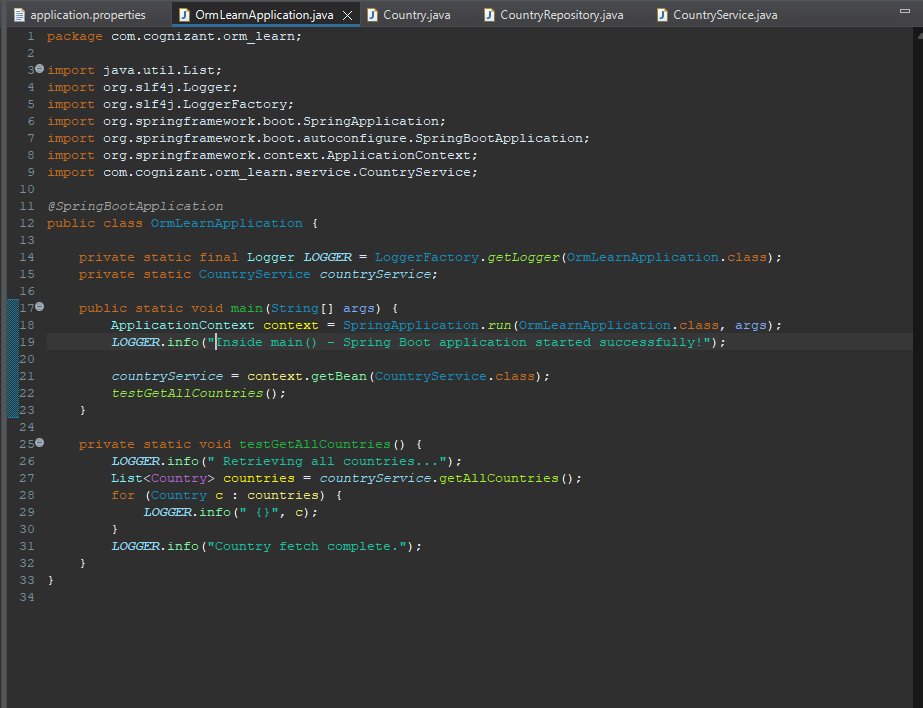
);

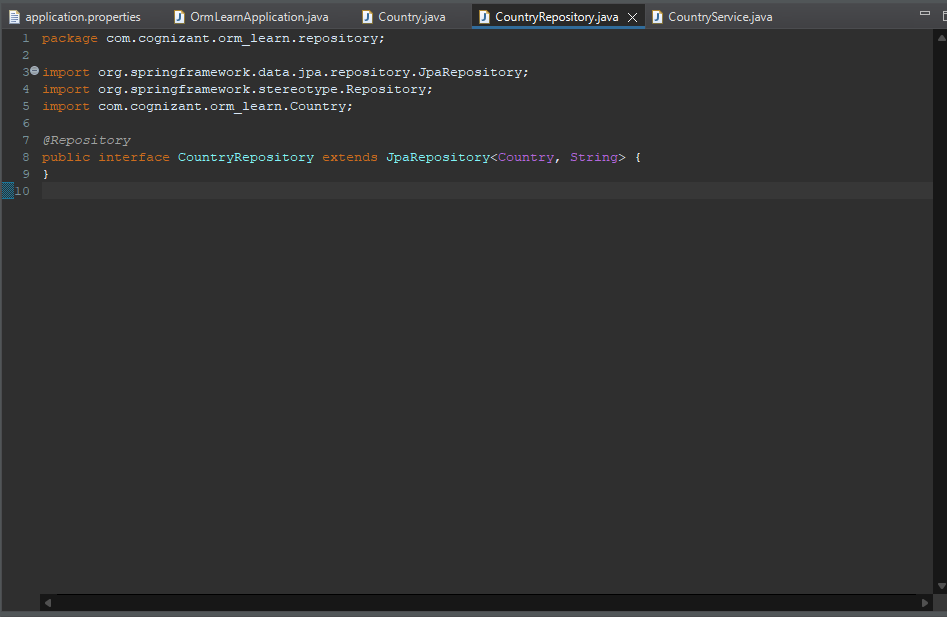
INSERT INTO country VALUES ('IN', 'India');

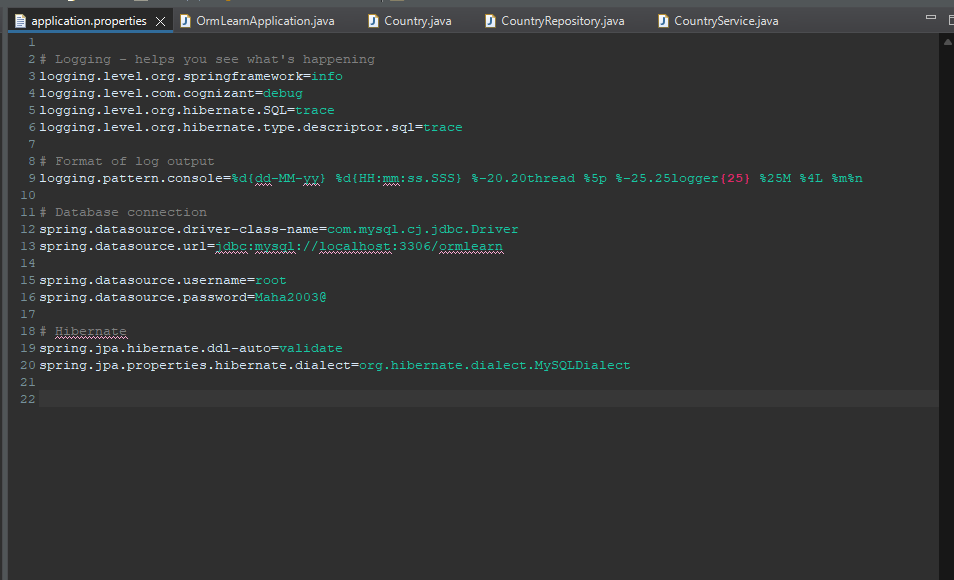
INSERT INTO country VALUES ('US', 'United States of America');

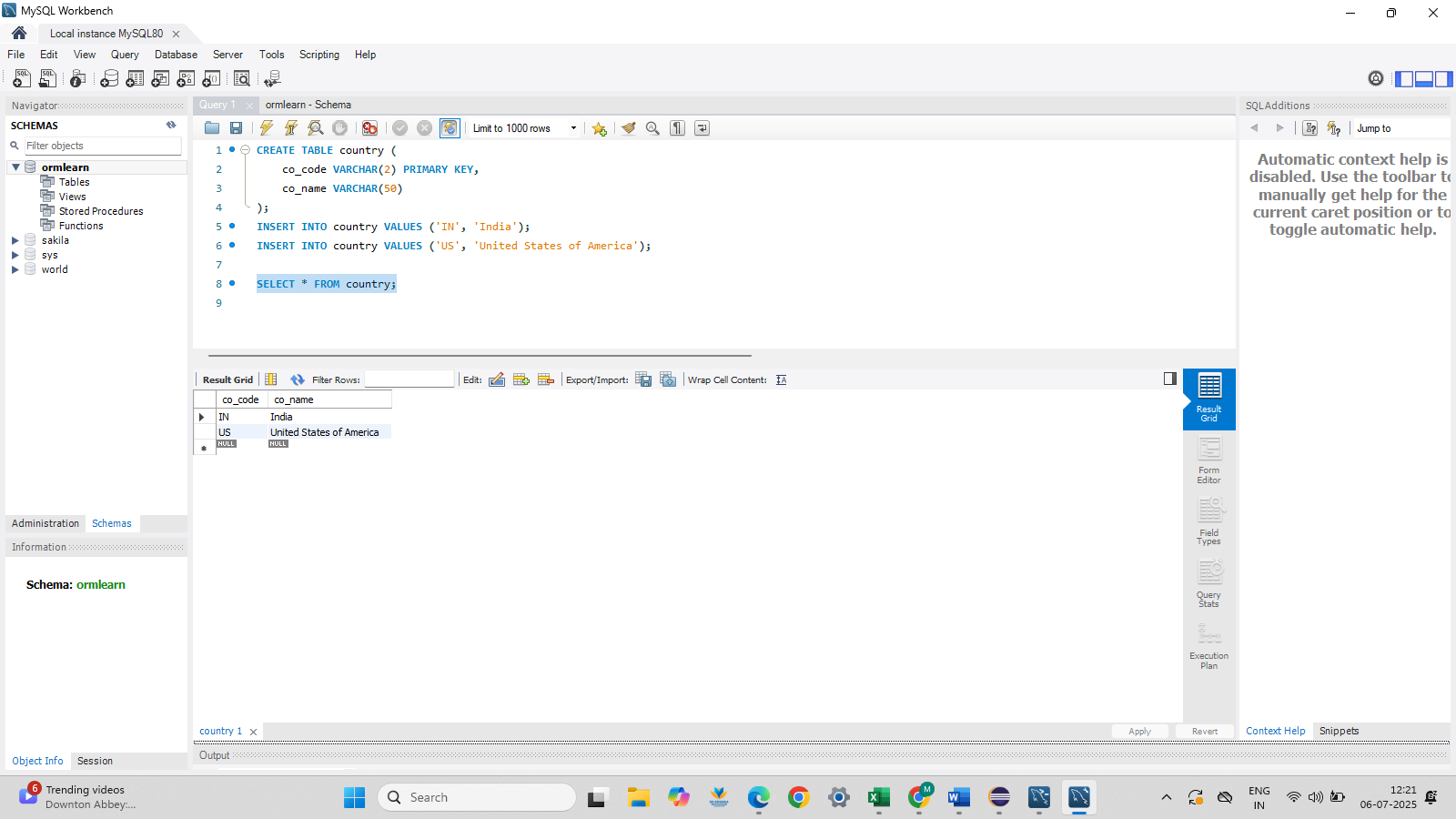
SELECT \* FROM country;

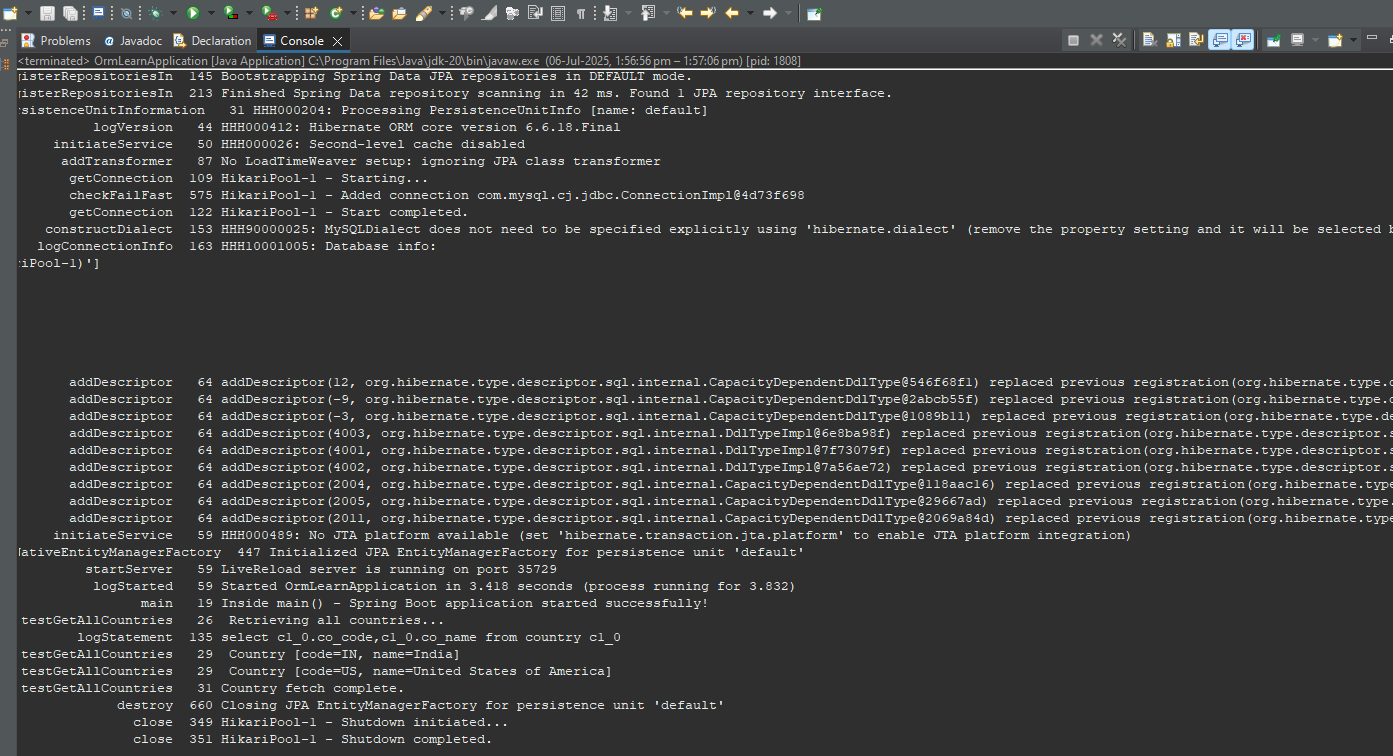
**SCREENSHOT**

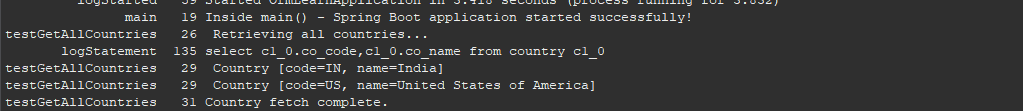










The console output confirms that the Spring Boot application was successfully connected to MySQL. It fetched records from the country table using JpaRepository without writing any SQL, demonstrating the power of Spring Data JPA abstraction over Hibernate and JPA.

**Conclusion**

This hands-on exercise helped me understand the working of Spring Boot, Hibernate (under the hood), and Spring Data JPA. I learned how to use annotations like @Entity, @Repository, and @Service to build a complete backend data access layer.

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

**Comparison Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature | JPA | Hibernate | Spring Data JPA |  |
| Type | Specification (API only) | ORM framework implementation | Spring module (abstraction over JPA) |  |
| Provides DB access? | No | Yes | Yes (via Hibernate or others) |  |
| Boilerplate Required | Yes | Yes | No (auto methods via interfaces) |  |
| Example Provider | — | Hibernate | Uses Hibernate under the hood |  |
| Managed By | Oracle (Java EE) | Red Hat | Spring Framework |  |

**Explanations**

**JPA (Java Persistence API)**

- JPA is just a specification that defines how Java objects are mapped to relational database tables.  
- It doesn't do any actual work itself.  
- It provides annotations likE @Entity, @Table, @Id, etc.

**Hibernate**

- Hibernate is the most popular implementation of JPA.  
- It performs the actual work of storing and retrieving Java objects from the database.  
- Hibernate adds additional features such as caching and HQL (Hibernate Query Language).

**Spring Data JPA**

- Spring Data JPA is a Spring module that abstracts and simplifies JPA-based data access.  
- It reduces boilerplate code by automatically implementing repository interfaces.  
- It allows developers to write repository interfaces like findByName() without writing the actual SQL.

**Code Examples**

JPA Entity Example

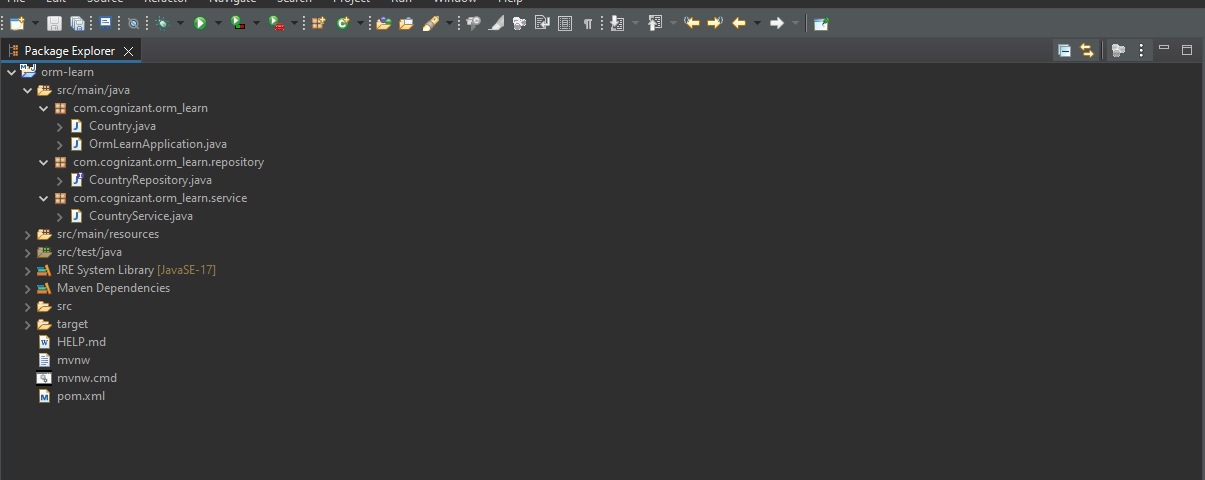
@Entity  
public class Employee {  
 @Id  
 private int id;  
 private String name;  
}

Hibernate Manual Example

*Session session = factory.openSession();  
Transaction tx = session.beginTransaction();  
session.save(new Employee());  
tx.commit();*

Spring Data JPA Example

*@Repository  
public interface EmployeeRepository extends JpaRepository<Employee, Integer> { }  
  
employeeRepository.save(new Employee());*

**SCREENSHOT **

Console and project structure showing Spring Data JPA running with Hibernate backend.

**Conclusion**

Spring Data JPA simplifies persistence logic by removing boilerplate code. It builds on top of JPA and Hibernate, allowing developers to focus on business logic instead of writing repetitive data access code.