**Hands on 1**

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

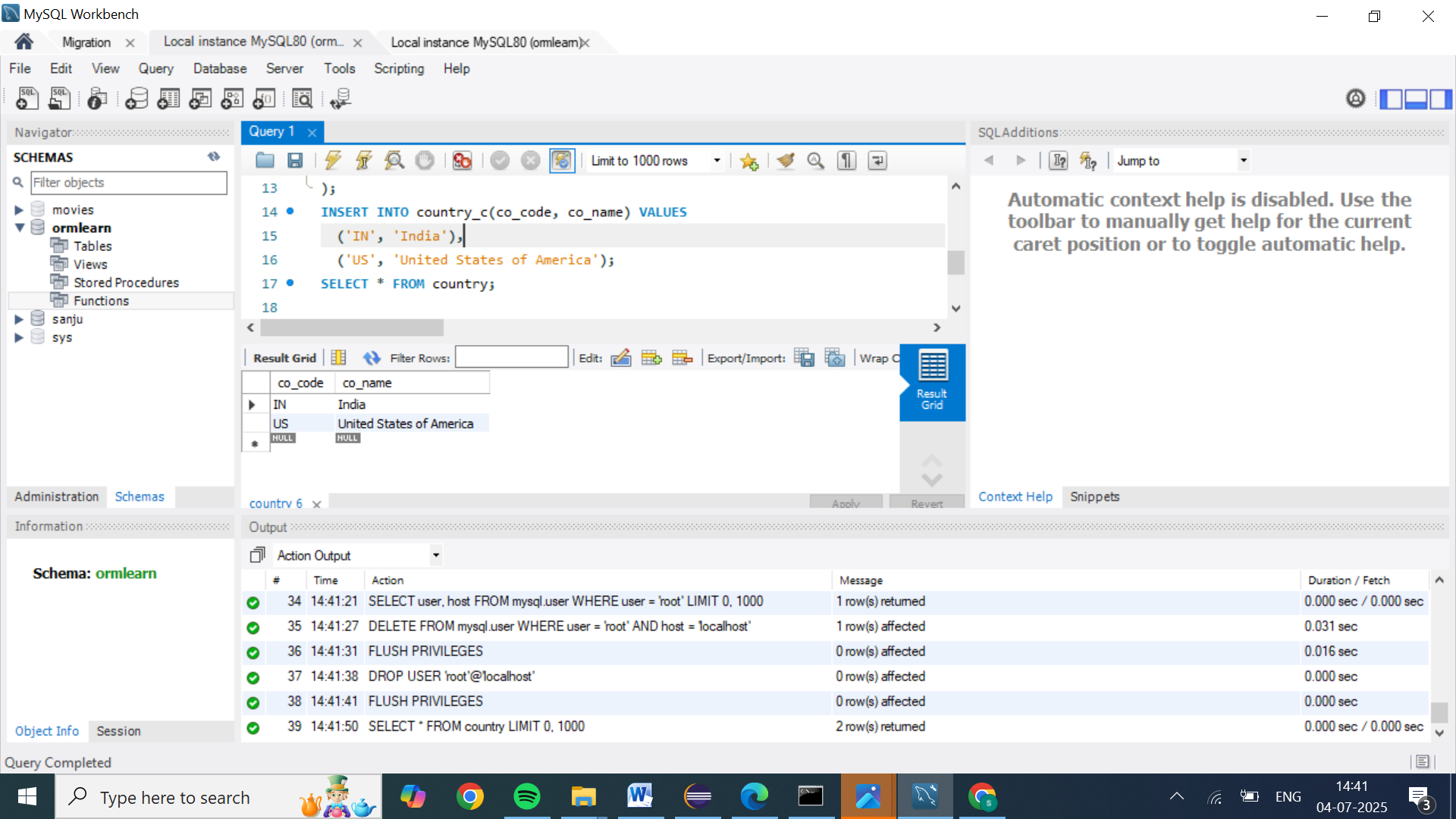
* MySQL Server 8.0
* MySQL Workbench 8
* Eclipse IDE for Enterprise Java Developers 2019-03 R
* Maven 3.6.2

**Create a Eclipse Project using Spring Initializr**

* Go to <https://start.spring.io/>
* Change Group as “com.cognizant”
* Change Artifact Id as “orm-learn”
* In Options > Description enter "Demo project for Spring Data JPA and Hibernate"
* Click on menu and select "Spring Boot DevTools", "Spring Data JPA" and "MySQL Driver"
* Click Generate and download the project as zip
* Extract the zip in root folder to Eclipse Workspace
* 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.

> mysql -u root -p

mysql> create schema ormlearn;



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

# Spring Framework and application log

logging.level.org.springframework=info

logging.level.com.cognizant=debug

# Hibernate logs for displaying executed SQL, input and output

logging.level.org.hibernate.SQL=trace

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

# Log pattern

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

# Database configuration

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

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

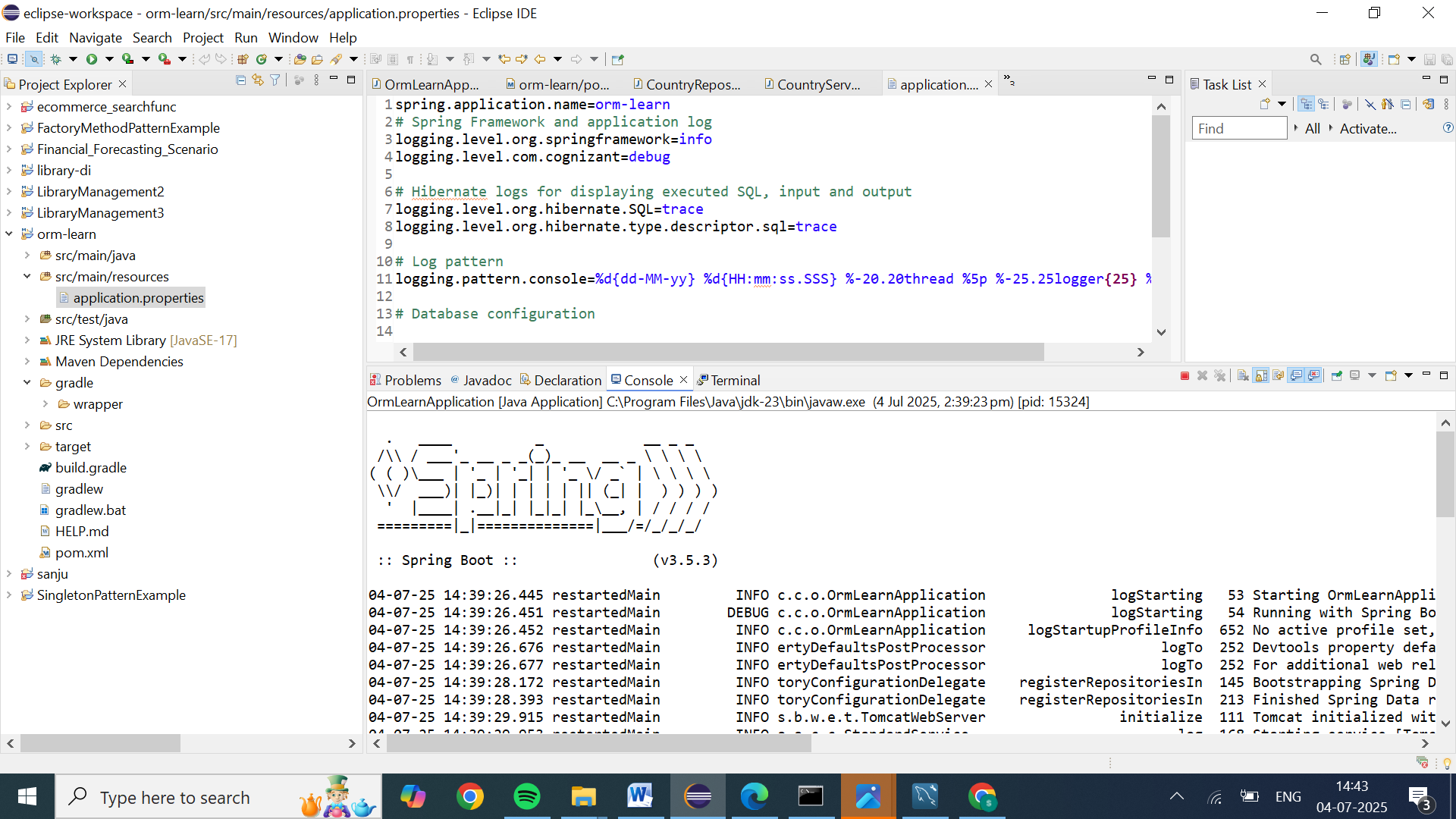
spring.datasource.username=root

spring.datasource.password=root

# Hibernate configuration

spring.jpa.hibernate.ddl-auto=validate

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



* Build the project using ‘mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com -Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456’ command in command line
* Include logs for verifying if main() method is called.

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

SME to walk through the following aspects related to the project created:

1. src/main/java - Folder with application code
2. src/main/resources - Folder for application configuration
3. src/test/java - Folder with code for testing the application
4. OrmLearnApplication.java - Walkthrough the main() method.
5. Purpose of @SpringBootApplication annotation
6. pom.xml
   1. Walkthrough all the configuration defined in XML file
   2. Open 'Dependency Hierarchy' and show the dependency tree.

**Country table creation**

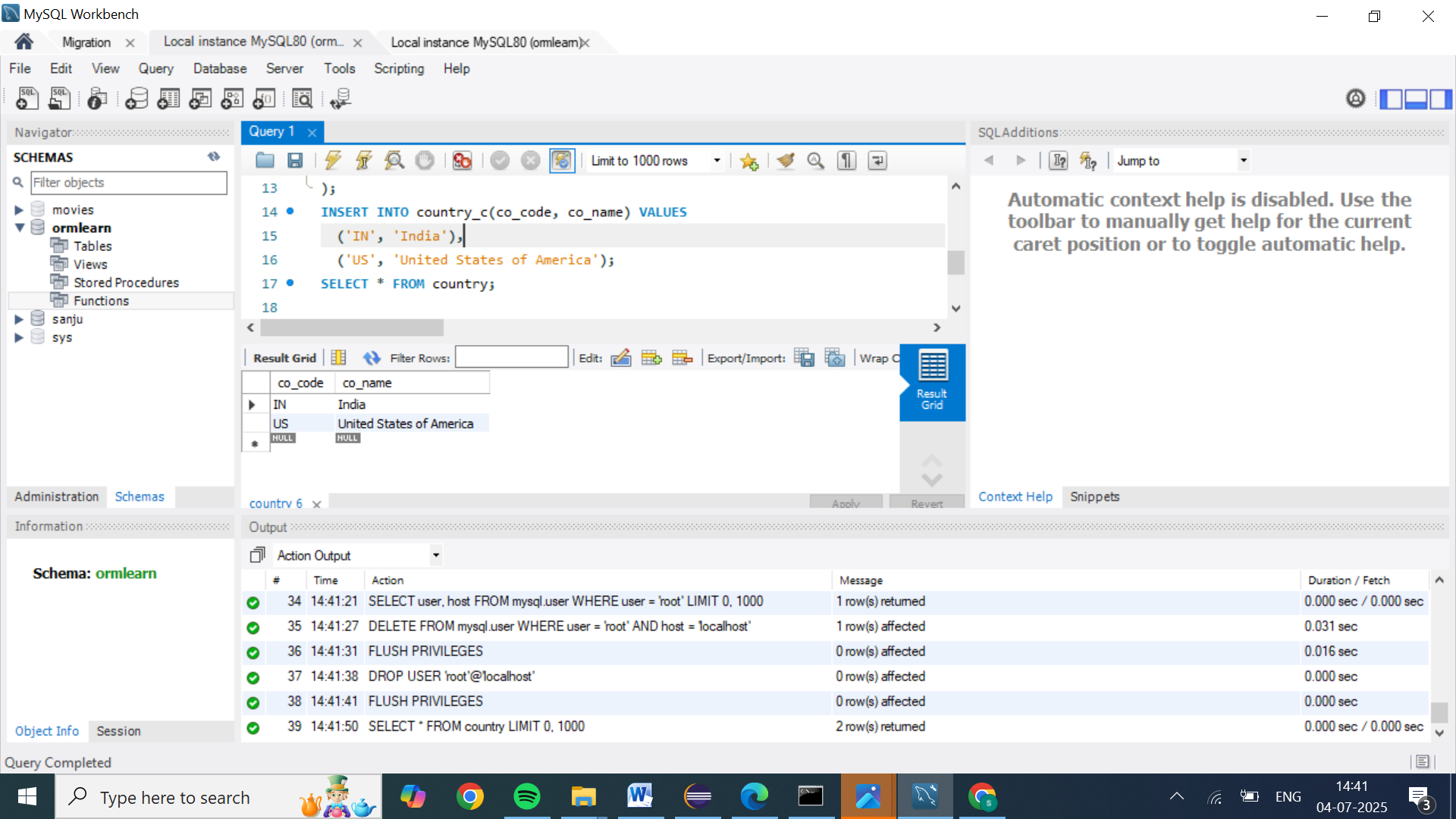
* Create a new table country with columns for code and name. For sample, let us insert one country with values 'IN' and 'India' in this table.

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');



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

* Open Eclipse with orm-learn project
* Create new package com.cognizant.orm-learn.model
* Create Country.java, then generate getters, setters and toString() methods.
* Include @Entity and @Table at class level
* Include @Column annotations in each getter method specifying the column name.

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name="country")

public class Country {

  @Id

    @Column(name="code")

    private String code;

    @Column(name="name")

    private String name;

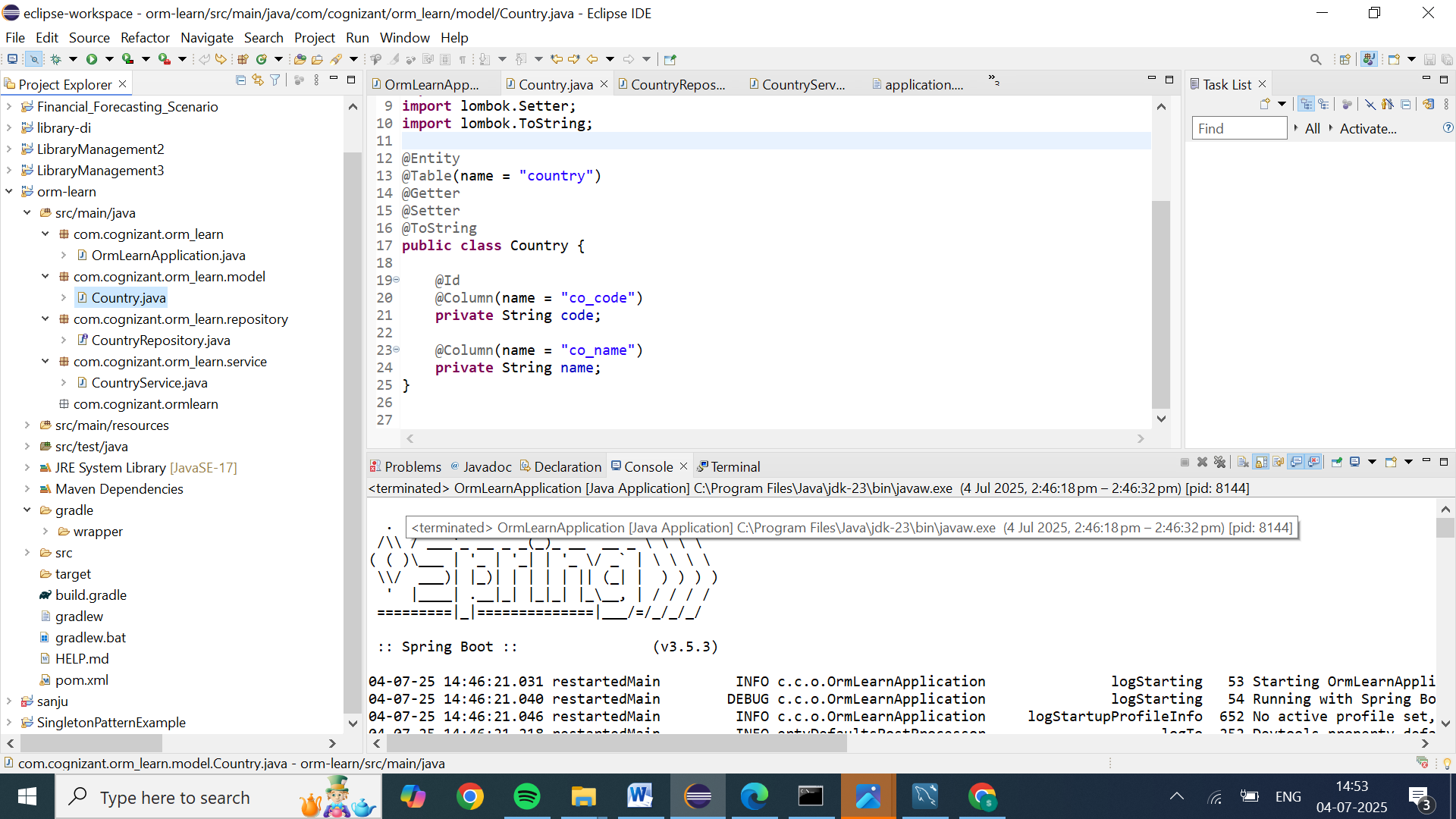
// getters and setters

  // toString()

}

*Notes:*

* @Entity is an indicator to Spring Data JPA that it is an entity class for the application
* @Table helps in defining the mapping database table
* @Id helps is defining the primary key
* @Column helps in defining the mapping table column



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

* Create new package com.cognizant.orm-learn.repository
* Create new interface named CountryRepository that extends JpaRepository<Country, String>
* Define @Repository annotation at class level

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

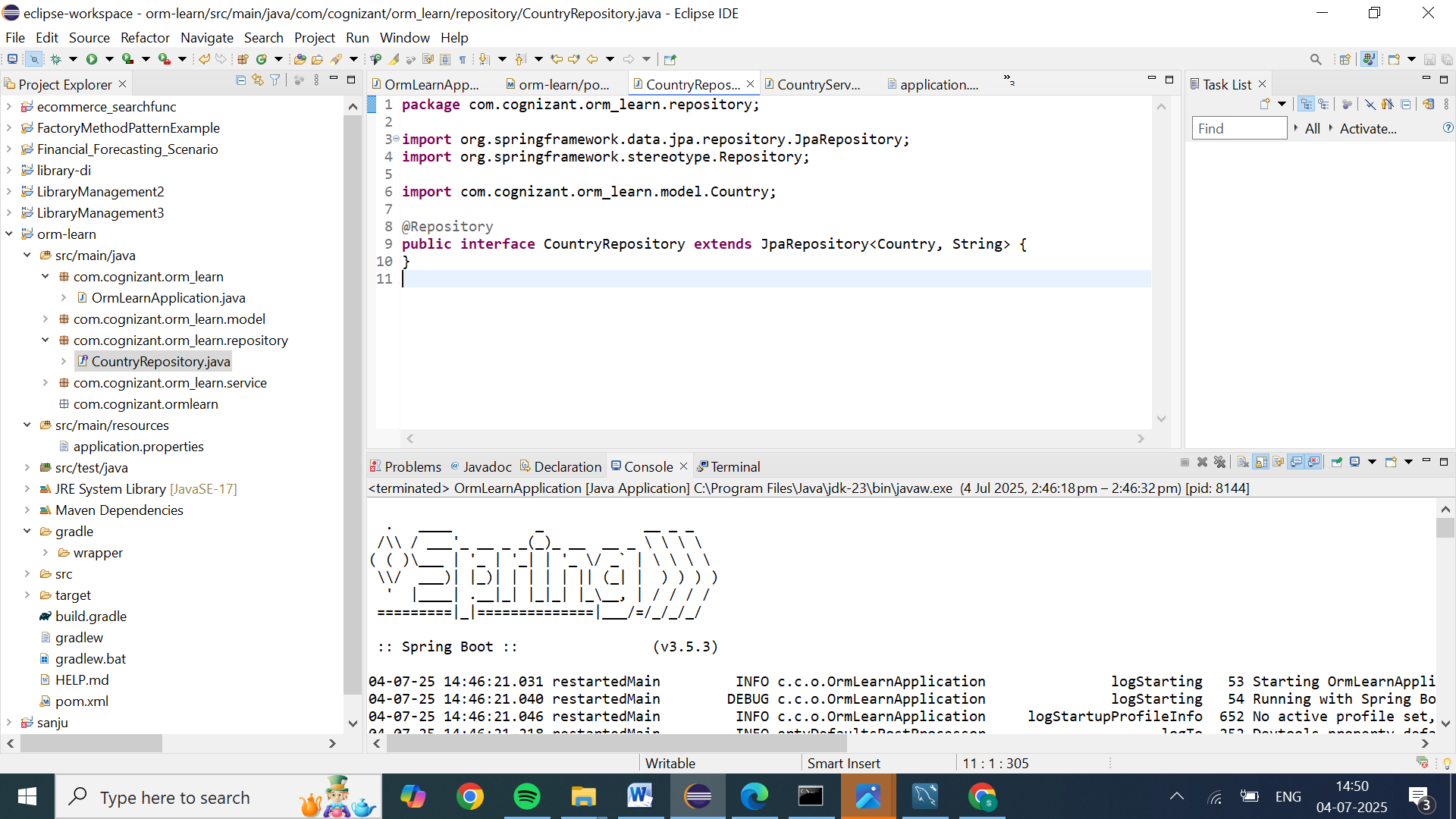
import org.springframework.stereotype.Repository;

import com.cognizant.ormlearn.model.Country;

@Repository

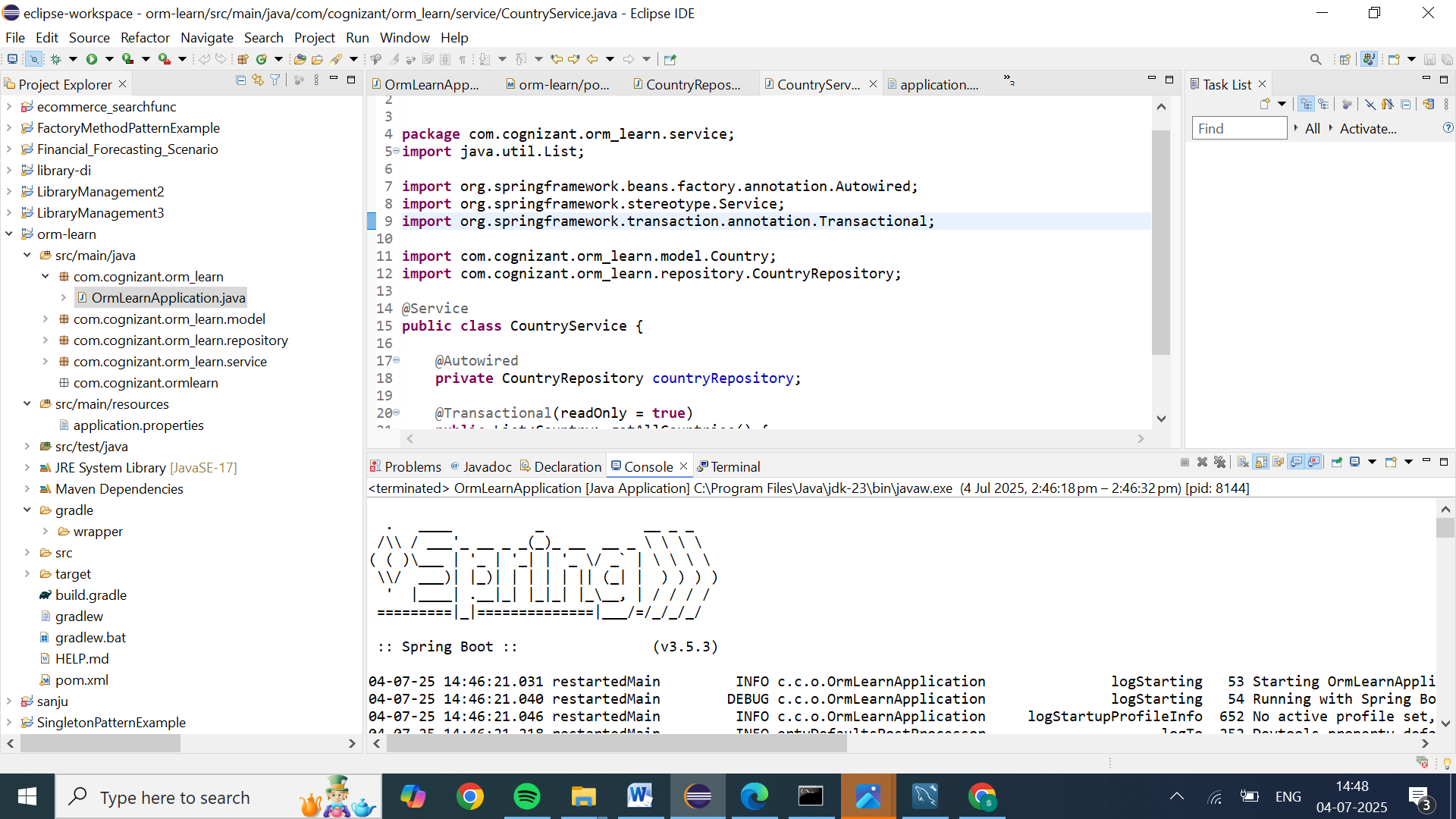
public interface CountryRepository extends JpaRepository<Country, String> {

}



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

* Create new package com.cognizant.orm-learn.service
* Create new class CountryService
* Include @Service annotation at class level
* Autowire CountryRepository in CountryService
* Include new method getAllCountries() method that returns a list of countries.
* Include @Transactional annotation for this method
* In getAllCountries() method invoke countryRepository.findAll() method and return the result



**Testing in OrmLearnApplication.java**

* Include a static reference to CountryService in OrmLearnApplication class

private static CountryService countryService;

* Define a test method to get all countries from service.

    private static void testGetAllCountries() {

        LOGGER.info("Start");

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

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

        LOGGER.info("End");

    }

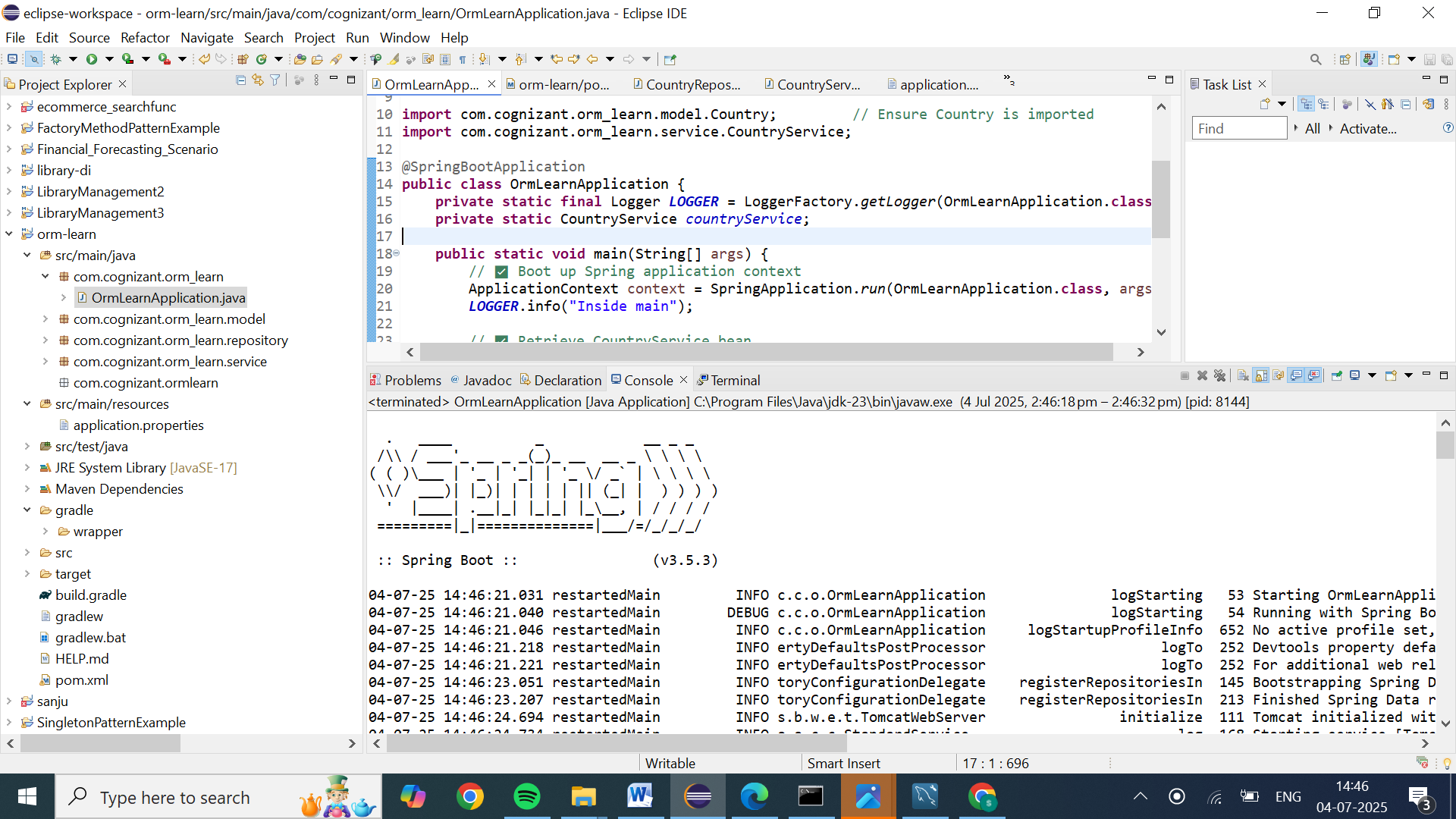
* Modify SpringApplication.run() invocation to set the application context and the CountryService reference from the application context.

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

        countryService = context.getBean(CountryService.class);

        testGetAllCountries();

* Execute main method to check if data from ormlearn database is retrieved.



OUTPUT information:

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v3.5.3)

:: Spring Boot :: (v3.5.3)

... (startup logs)

INFO o.h.j.i.util.LogHelper - HHH000204: Processing PersistenceUnitInfo [name: default]

INFO org.hibernate.Version - HHH000412: Hibernate ORM core version 6.6.18.Final

INFO c.z.h.HikariDataSource - HikariPool-1 - Starting...

INFO c.z.h.pool.HikariPool - Added connection com.mysql.cj.jdbc.ConnectionImpl@xxxxxx

INFO c.z.h.HikariDataSource - HikariPool-1 - Start completed.

...

INFO o.s.b.w.e.t.TomcatWebServer - Tomcat started on port 8080 (http) with context path '/'

INFO c.c.o.OrmLearnApplication - Started OrmLearnApplication in X.XXX seconds

INFO c.c.o.OrmLearnApplication - Inside main

INFO c.c.o.OrmLearnApplication - Start

DEBUG org.hibernate.SQL - select c1\_0.co\_code,c1\_0.co\_name from country c1\_0

DEBUG c.c.o.OrmLearnApplication - countries=[com.cognizant.orm\_learn.model.Country@519da01b, com.cognizant.orm\_learn.model.Country@3d65281e]

INFO c.c.o.OrmLearnApplication - End

Explain the difference between Java Persistence API, Hibernate and Spring Data JPA

* + JPA (Java Persistence API), JPA is a specification (JSR 338), JPA does not have implementation, Hibernate is one of the implementation for JPA, Hibernate is a ORM tool, Spring Data JPA is an abstraction above Hibernate to remove boiler plate code when persisting data using Hibernate.
    - Difference between Spring Data JPA and Hibernate - https://dzone.com/articles/what-is-the-difference-between-hibernate-and-sprin-1
    - Intro to JPA - https://www.javaworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html

**1. JPA (Java Persistence API)**

* **What it is**  
  A formal specification (JSR‑338, now Jakarta Persistence) that defines how to map Java objects to relational DB tables. It provides interfaces like EntityManager, CriteriaBuilder, and the JPQL query language—but *doesn’t execute any persistence operations itself*.
* **Implementation-agnostic**  
  Requires a provider—such as Hibernate, EclipseLink, or OpenJPA—to do the actual work
* **When to use**  
  Great for developers aiming for portable, JPA-compliant codebases, but involves manually writing DAOs, transaction handling, and JPQL/Criteria queries.
* The Java Persistence API provides a specification for persisting, reading, and managing data from your Java object to relational tables in the database.
* Read more about JPA at [JPA Tutorial - Java Persistence API](http://www.javaguides.net/p/jpa-tutorial-java-persistence-api.html) (you will learn everything about JPA here).

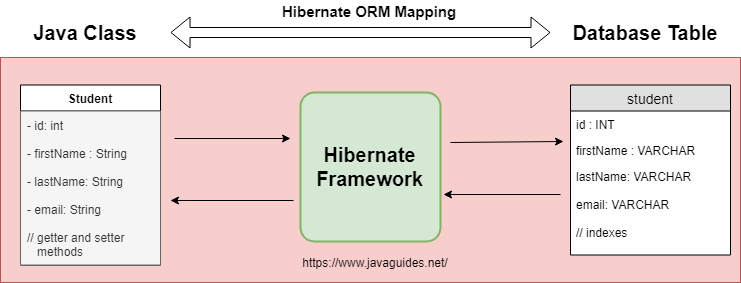
**2. Hibernate**

* **What it is**  
  A robust, full-featured ORM framework that implements JPA (though it can also be used standalone)
* **Feature highlights**
  + Maps POJOs to DB tables, offering lazy loading, interceptors, and rich caching Supports HQL, JPQL, Criteria API, and native SQL
  + Includes first- and second-level caches, plus transaction/session management—works both with and without Spring.
* **When to use**  
  Ideal when you need low-level SQL control, performance optimization, or are building outside of the Spring ecosystem.

Hibernate is an object-relational mapping solution for Java environments. Object-relational mapping or ORM is the programming technique to map application domain model objects to the relational database tables.

Hibernate provides a reference implementation of the Java Persistence API that makes it a great choice as an ORM tool with the benefits of loose coupling.

**Example**: Below diagram shows an *Object Relational Mapping* between the **Student** Java class and the **students**tablein the database.



* **Note that JPA is a specification and Hibernate is a JPA provider or implementation**

**3. Spring Data JPA**

* **What it is**  
  A Spring-based abstraction layer that builds on JPA (typically using Hibernate under the covers), simplifying data-access code
* **Key benefits**
  + Repository interfaces like JpaRepository auto-generate CRUD, pagination, sorting, and query methods based on naming conventions
  + Tight integration with Spring’s @Transactional, dependency injection, and JpaSpecificationExecutor for advanced queries
  + Supports derived queries, @Query, Criteria API, and Specifications without manual implementation
  + Requires a JPA provider (commonly Hibernate) to function
* **When to use**  
  Perfect for Spring Boot projects focused on rapid development and minimal boilerplate, unless you need very fine-grained ORM control.
* Spring Data is a part of the [Spring Framework](http://www.javaguides.net/p/spring-framework.html). The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.
* Spring Data JPA is not a JPA provider. It is a library/framework that adds an extra layer of abstraction on the top of our JPA provider (like Hibernate).
* Read more about Spring Data JPA at [Spring Data JPA Tutorial](http://www.javaguides.net/p/spring-data-jpa-tutorial.html) (you will learn everything about Spring Data JPA here).
* Now, you are familiar with the Definition of JPA, Hibernate, and Spring Data JPA. Now, let's discuss the difference between Hibernate and Spring Data JPA.

**🆚 Quick Comparison**

| **Aspect** | **JPA** | **Hibernate** | **Spring Data JPA** |
| --- | --- | --- | --- |
| **Nature** | Spec + APIs + JPQL/Criteria | ORM implementation + rich features | Repository abstraction layer |
| **Provider** | None | Hibernate (or others) | Requires JPA provider (e.g. Hibernate) |
| **Query options** | JPQL, Criteria API | HQL, JPQL, Criteria, native SQL | Method-derived, @Query, JPQL, Criteria |
| **Boilerplate** | High — write DAOs & transactions | Moderate — manage sessions manually | Low — repository methods auto-implemented |
| **Caching** | Basic JPA-level caching | Advanced first/second-level caching | Inherits provider’s caching capabilities |
| **Ideal for** | Portable, non-Spring, spec-compliant apps | Performance-heavy or non-Spring use | Fast Spring Boot CRUD-centric development |

Spring Data JPA is not an implementation or JPA provider, it's just an abstraction used to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.

I hope this article is useful to you. You can find more answers to this question at [StackOverflow](https://stackoverflow.com/questions/23862994/what-is-the-difference-between-hibernate-and-spring-data-jpa/23863416" \l "23863416" \t "_blank).

Hibernate provides a reference implementation of the Java Persistence API that makes it a great choice as an ORM tool with the benefits of loose coupling.

Remember, Spring Data JPA always requires the JPA provider such as Hibernate or Eclipse Link.

**Final Takeaway**

* **JPA** = the rulebook—interface definitions and query standards.
* **Hibernate** = the engine—implements JPA and adds powerful ORM capabilities.
* **Spring Data JPA** = the convenience layer—packs JPA/Hibernate into clean, easy-to-use repositories for Spring projects.