# **WEEK-03**

# **HANDS ON SOLUTIONS**

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

**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.

**SOLUTION:**

In this hands-on, I developed a Spring Boot application that demonstrates the use of Spring Data JPA and Hibernate for interacting with a relational database.

The application connects to a MySQL database and is designed to retrieve country data from a table using repository abstraction.

The objective was to gain a clear understanding of how to configure JPA entities, set up a repository layer, and manage service-level business logic using Spring's layered architecture.

The project was set up using Spring Initializr with key dependencies such as Spring Boot DevTools, Spring Data JPA, and MySQL Driver. Once generated, the Maven project was imported into Eclipse.

All database configurations, including driver, URL, credentials, and Hibernate dialect, were defined in the application.properties file.

Logging settings were also configured to monitor application behavior, Spring framework events, and SQL query execution through Hibernate’s SQL tracing capabilities.

The application models a Country entity that represents a table in the MySQL database. This entity class was annotated to map Java fields to corresponding table columns and define the primary key.

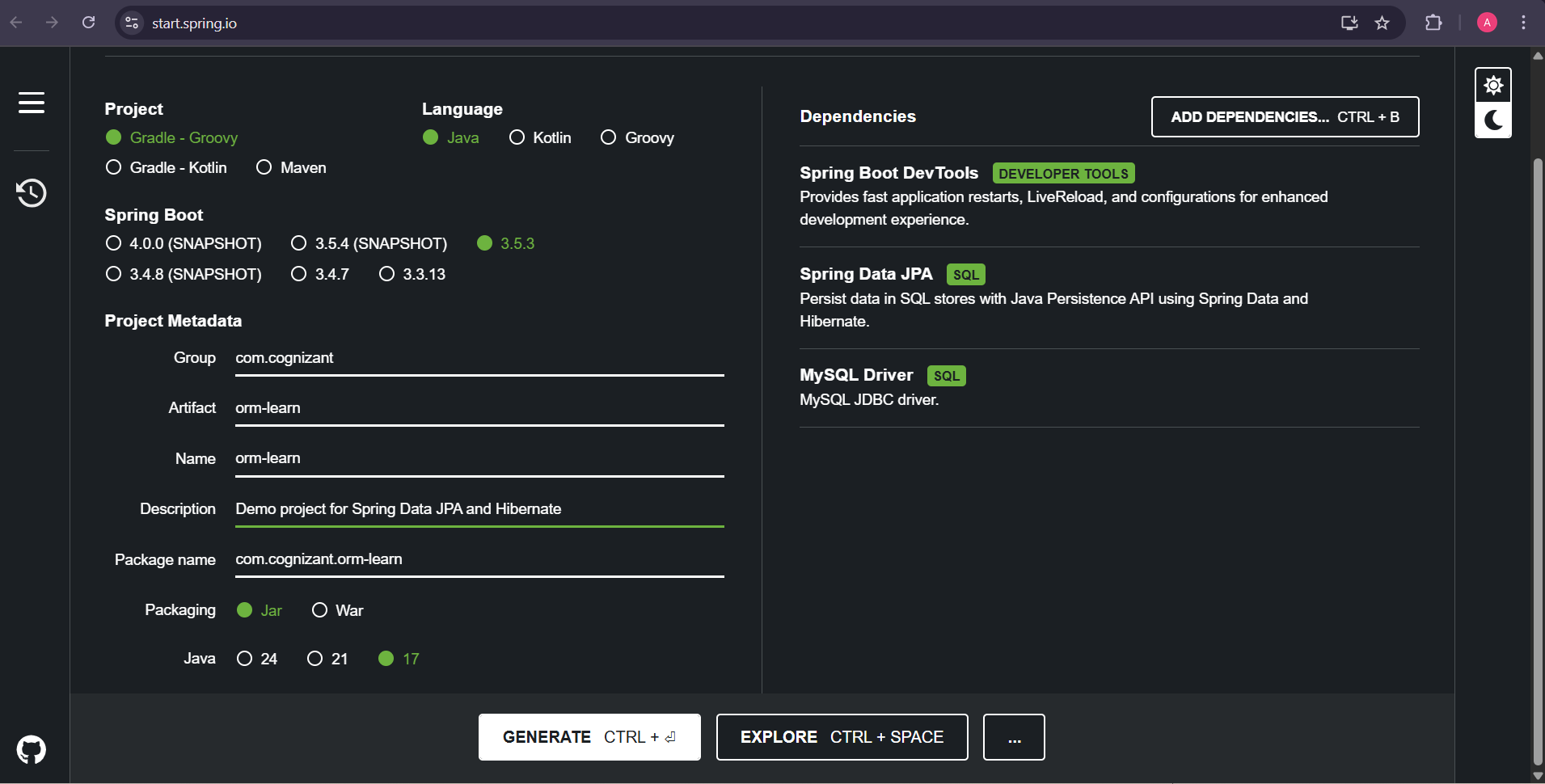
A repository interface was created by extending Spring Data JPA's JpaRepository, providing access to built-in methods for data retrieval without requiring custom SQL or boilerplate code.

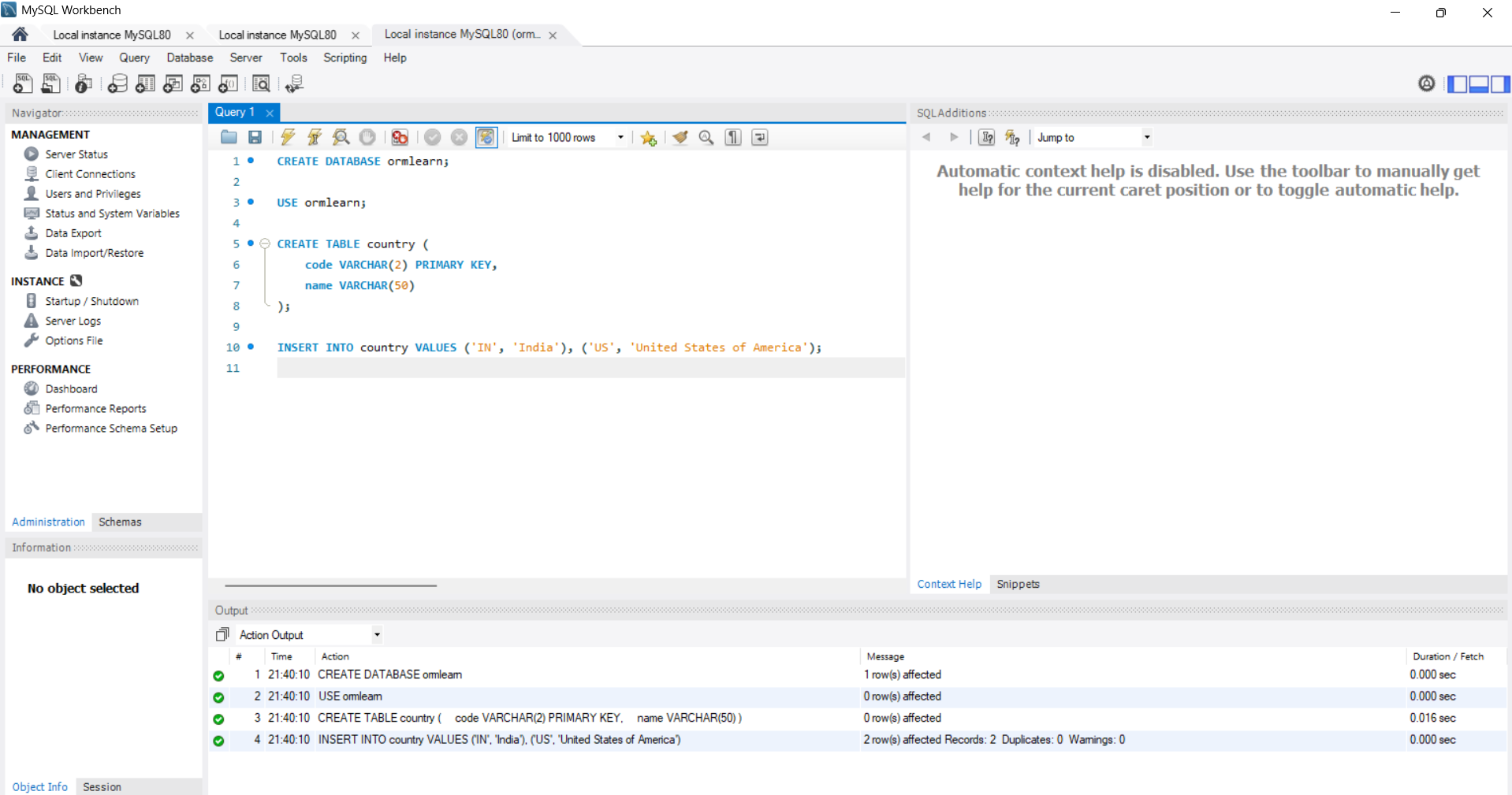
A service class was implemented to encapsulate the business logic of fetching country records from the database.

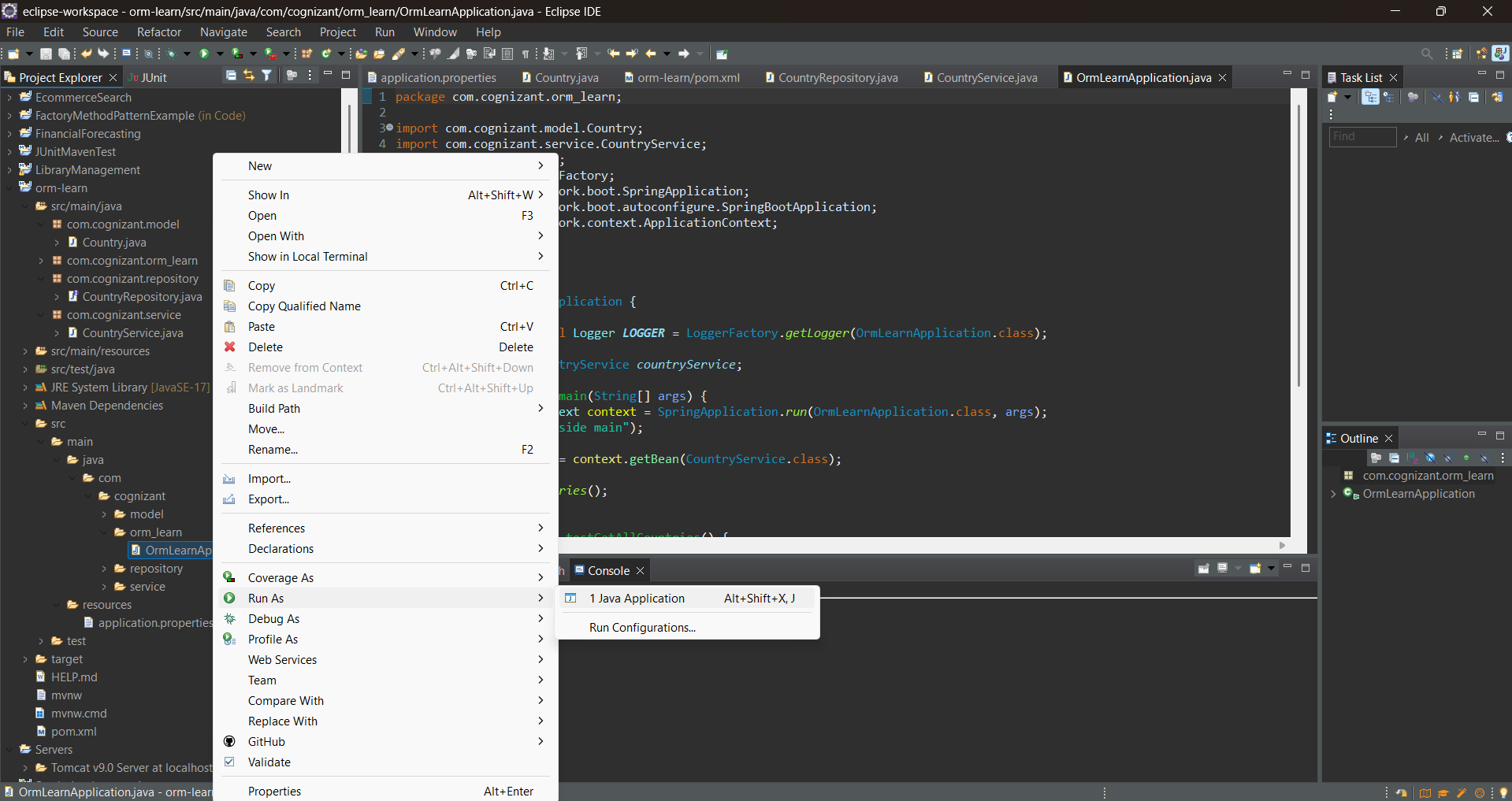
This service class was annotated for dependency injection and transaction management, enabling smooth interaction with the repository layer. The main class of the application was used to bootstrap the Spring context, retrieve the service bean, and invoke a method to list all countries from the database. Logging was used extensively to trace each phase of the application execution, including the invocation of the main method and the results retrieved from the database.

The database schema was created manually in MySQL and populated with sample data to verify the application's functionality. Upon running the application, the Spring Boot logs confirmed successful connection to the database, proper initialization of the persistence context, and the retrieval of country data, which was logged as output.

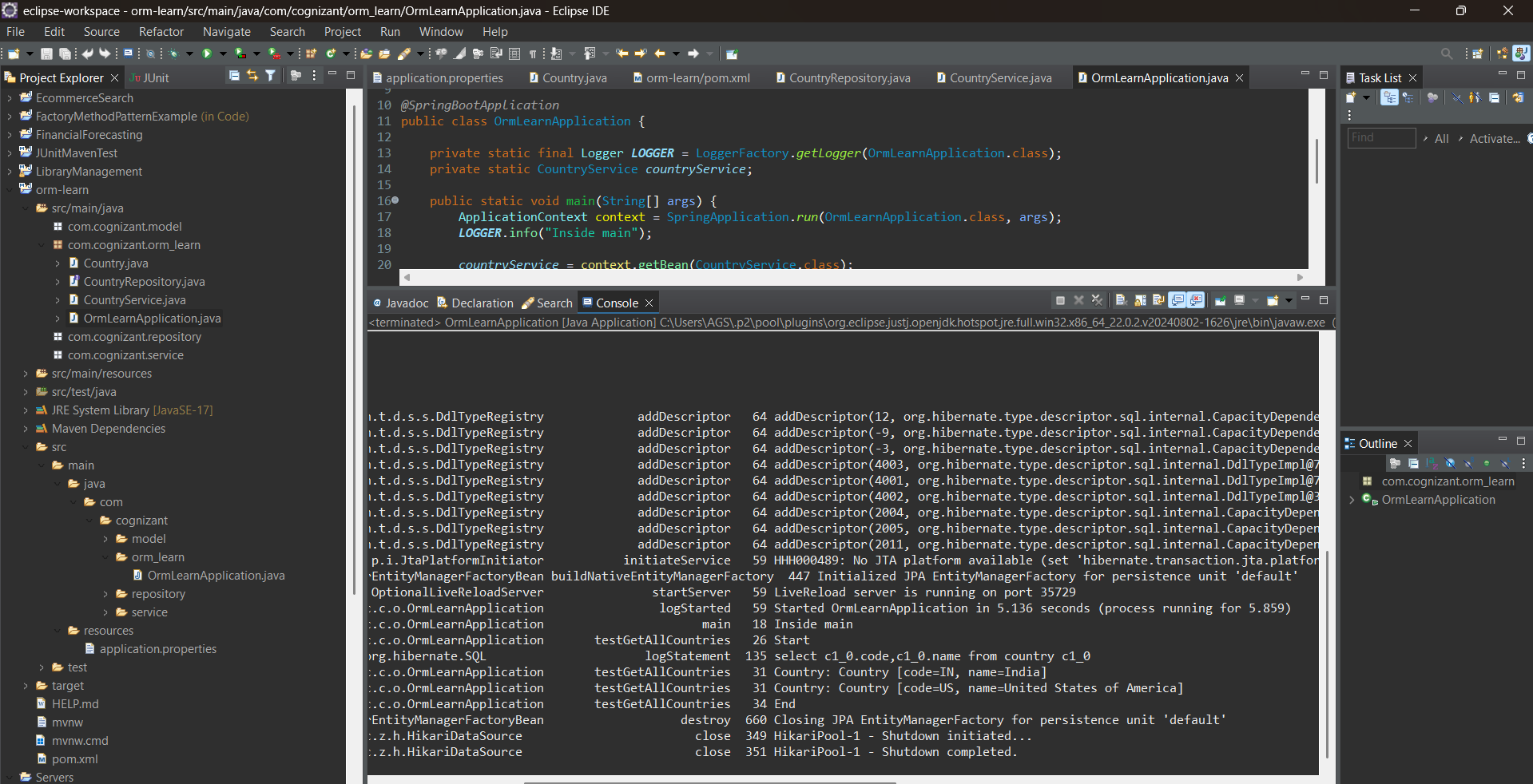
This exercise successfully demonstrated the use of Spring Boot and JPA for building a lightweight and efficient data access layer with minimal configuration and maximum clarity.







**OUTPUT:**



**EXERCISE-02**

**Difference between JPA, Hibernate and Spring Data JPA Java Persistence API (JPA)**

· JSR 338 Specification for persisting, reading and managing data from Java objects

· Does not contain concrete implementation of the specification

· Hibernate is one of the implementation of JPA

Hibernate

· ORM Tool that implements JPA

Spring Data JPA

· Does not have JPA implementation, but reduces boiler plate code

· This is another level of abstraction over JPA implementation provider like Hibernate

· Manages transactions

Refer code snippets below on how the code compares between Hibernate and Spring Data JPA Hibernate

/\* Method to CREATE an employee in the database \*/

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 EmployeeRespository.java

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

EmployeeService.java

@Autowire

private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

employeeRepository.save(employee);

}

SOLUTION:

1. JPA (Java Persistence API) JPA is a standard specification (JSR 338) for object-relational mapping (ORM) in Java. It provides APIs and annotations to manage relational data using Java objects. Important point: JPA is just an interface — it doesn't provide actual implementation. You need a JPA provider (like Hibernate) to make it work.
2. Hibernate Hibernate is the most popular implementation of JPA. It is a fully featured ORM tool that: Implements all JPA annotations and interfaces.

Provides additional features like caching, lazy loading, batch fetching, etc. You still need to write boilerplate code (e.g., sessions, transactions, etc.) when using Hibernate directly.

1. Spring Data JPA Spring Data JPA is not an implementation of JPA, but a Spring-based abstraction on top of JPA (and typically Hibernate).

It reduces boilerplate code by: Auto-creating repository methods (save(), findById(), deleteById(), etc.) Managing transactions internally Handling EntityManager sessions for you You just define interfaces, and Spring generates the code at runtime.

Spring Data JPA makes working with JPA + Hibernate much simpler by auto-wiring most of the plumbing.

This hands-on project demonstrates the use of Spring Boot, Spring Data JPA, and Hibernate for implementing a simple employee management system backed by a MySQL database.

The goal is to show how Spring Data JPA abstracts away boilerplate code compared to traditional Hibernate usage.

1. Application Architecture The project follows a standard layered structure-

Model Layer: Defines the Employee entity annotated with @Entity, mapped to the employee table with fields like id, name, department, and salary, using @Id and @Column.

Repository Layer: EmployeeRepository extends JpaRepository<Employee, Integer>, giving access to built-in CRUD operations without writing boilerplate queries. Service Layer: EmployeeService is annotated with @Service, handles business logic, and autowires the repository. It includes methods like addEmployee() and getAllEmployees() with @Transactional for data integrity.

Main Application: EmployeeAppApplication uses ApplicationContext to retrieve the service bean and tests methods for adding and retrieving employee records.

1. Spring Boot & JPA Integration JPA Implementation: Hibernate is used as the JPA provider.

Configuration: Done via application.properties, including JDBC URL, driver, dialect, and logging levels.

Transaction Management: Handled automatically by Spring using @Transactional.

Repository Scanning: Spring scans for @Repository interfaces, auto-generates proxies for data access.

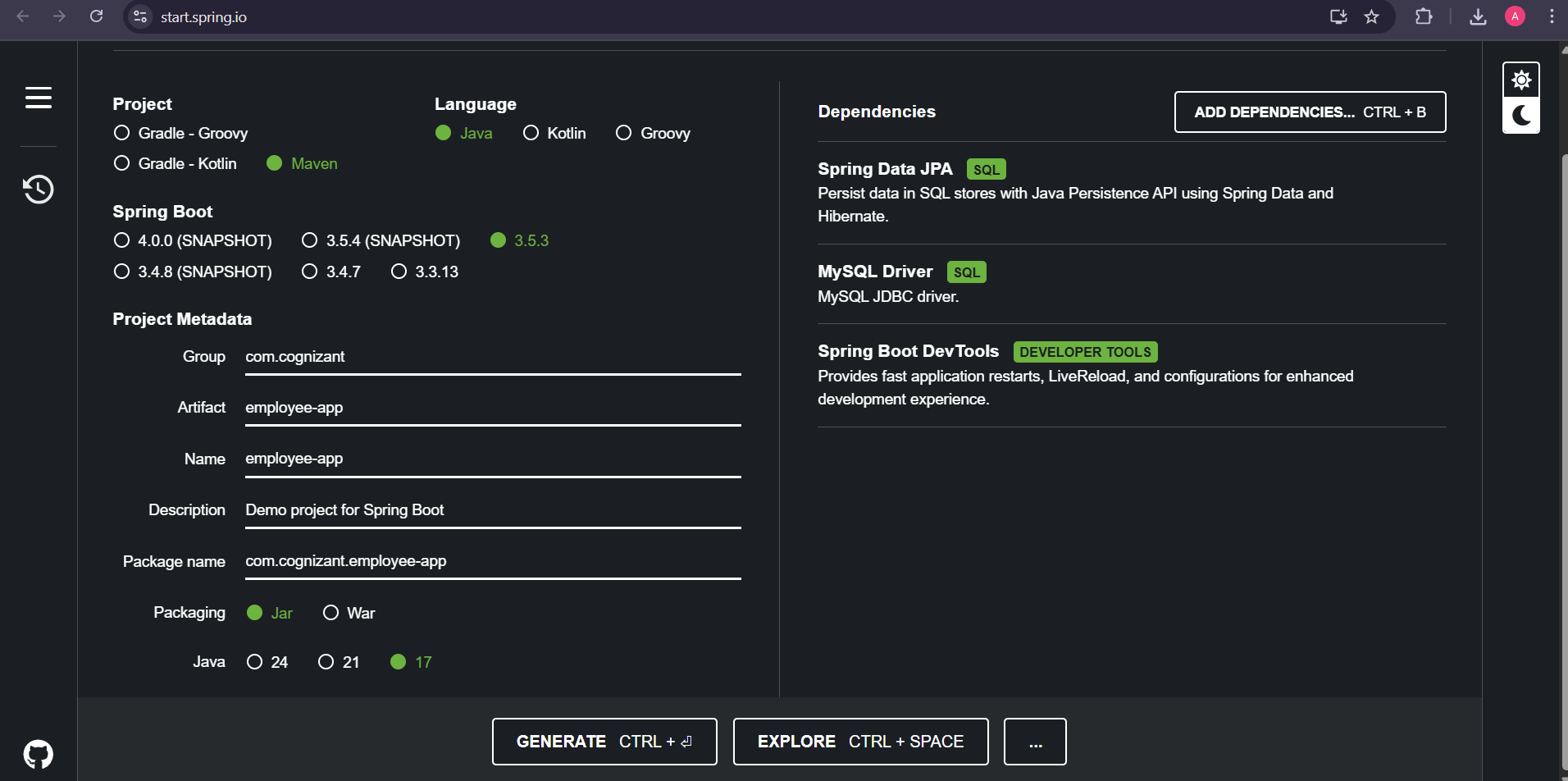
1. MySQL Integration The app connects to a MySQL 8.0+ database using mysql-connector-j driver. HikariCP is used for efficient connection pooling. Data is persisted and retrieved from the employee table.
2. Logging and Debugging Custom logging is enabled to capture Spring lifecycle (info), Hibernate SQL execution (trace), and application-specific debug logs. SQL queries (e.g., INSERT, SELECT) are visible in the console for verification.
3. Advantages of Spring Data JPA- Removes the need for manual session handling, transaction management, and query construction.

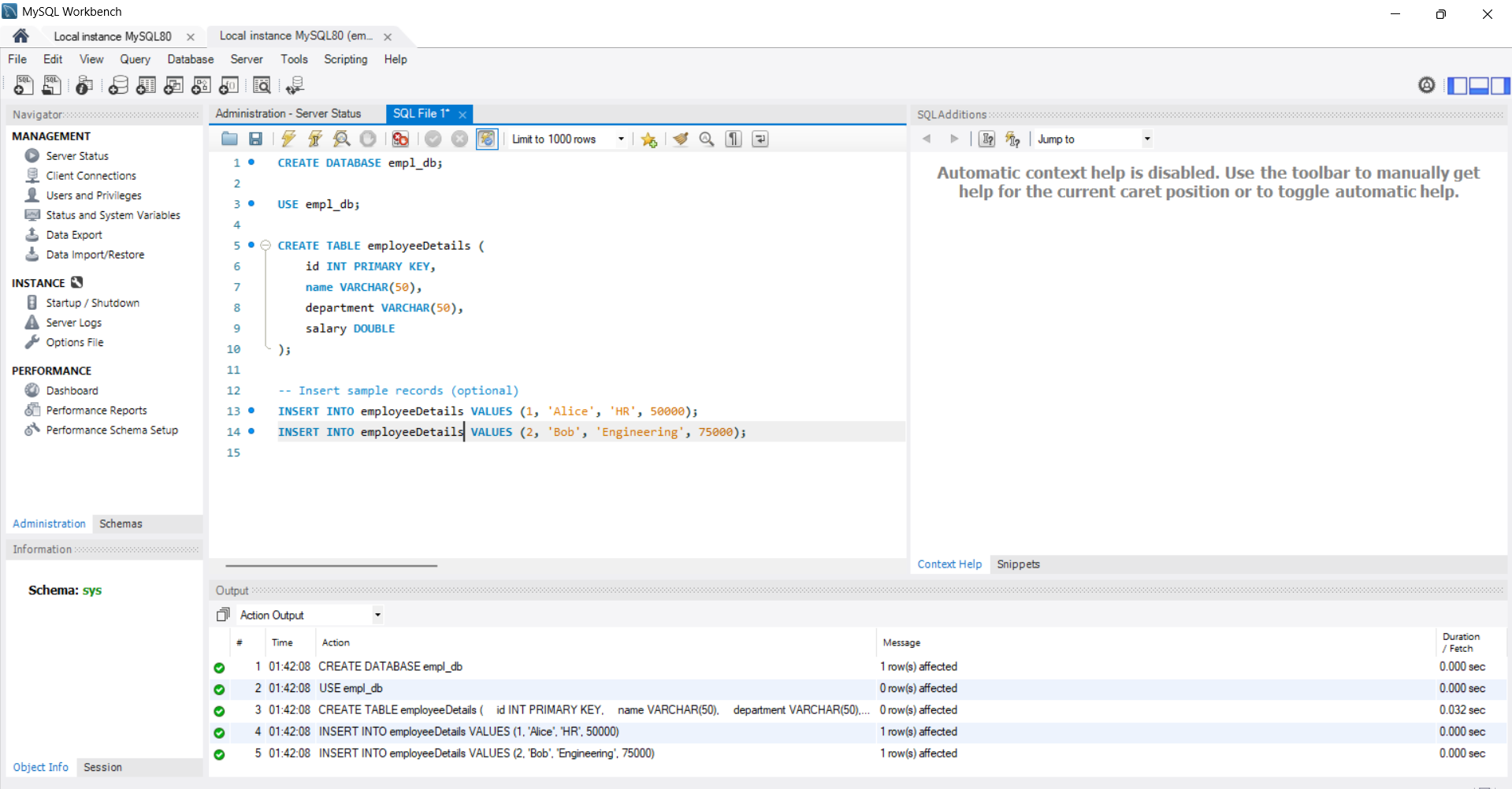
Reduces boilerplate compared to traditional Hibernate: No need to write session.save(), transaction rollbacks, or session closures. Provides built-in methods like save(), findAll(), deleteById().

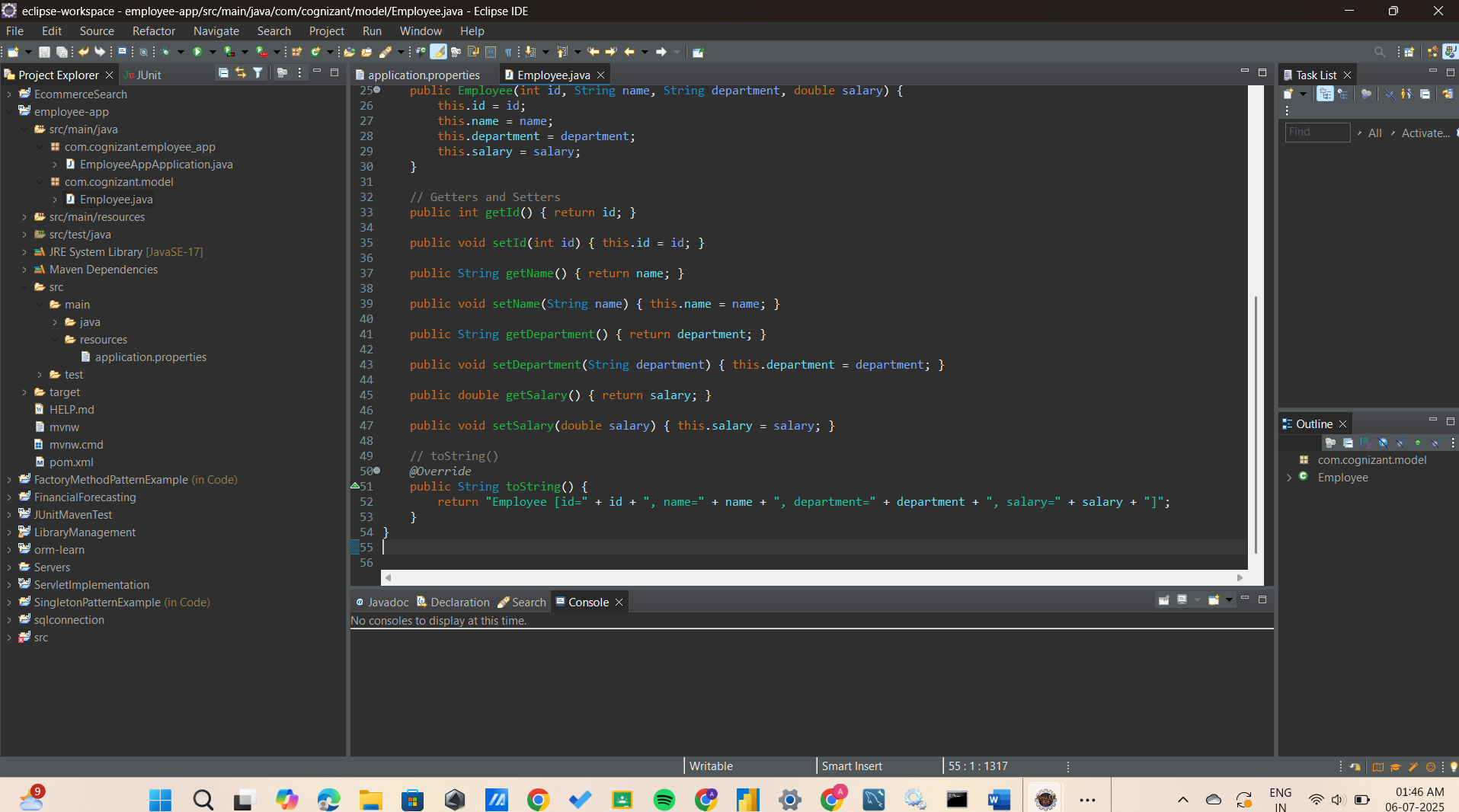
The Spring Boot application using Spring Data JPA was executed successfully, demonstrating seamless integration with Hibernate as the JPA provider and MySQL as the database.

The EmployeeService leveraged the EmployeeRepository to perform CRUD operations with minimal boilerplate code, highlighting the abstraction benefits of Spring Data JPA.

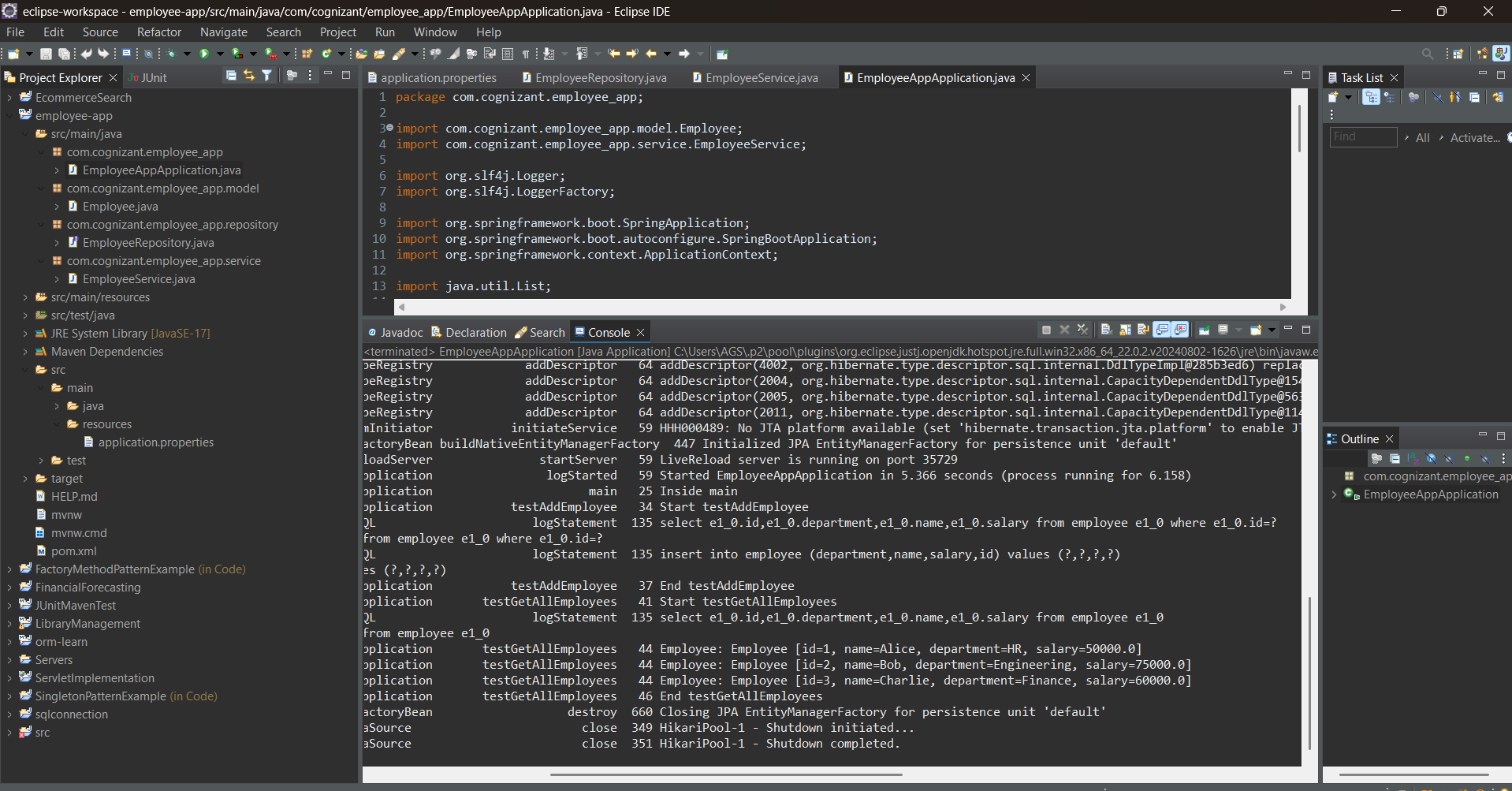
The application logged SQL queries and transaction activities, confirming proper entity mapping and data persistence. Employee records were inserted and retrieved correctly, with output showing the expected employee details, validating the end-to-end functionality from the service layer to the database.







**OUTPUT:**

****