**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.

**APPROACH**

* Create a Maven project and add spring-context dependency in pom.xml.
* Create an applicationContext.xml in src/main/resources and define beans.
* Create BookRepository and BookService with a setter injection.
* Write a LibraryApp class to load the Spring context and retrieve the bookService bean.
* Run the app and see the output in the console.

**CODE**

**pom.xml**

**Added following dependencies in pom.xml**

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.32</version> <!-- Latest stable Spring 5 -->

</dependency>

</dependencies>

**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 String getBookDetails() {

return "Spring in Action - Craig Walls";

}

}

**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 displayBook() {

System.out.println("Book: " + bookRepository.getBookDetails());

}

}

**LibraryApp.java**

package com.library1;

import com.library.service.BookService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LibraryApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

BookService bookService = context.getBean("bookService", BookService.class);

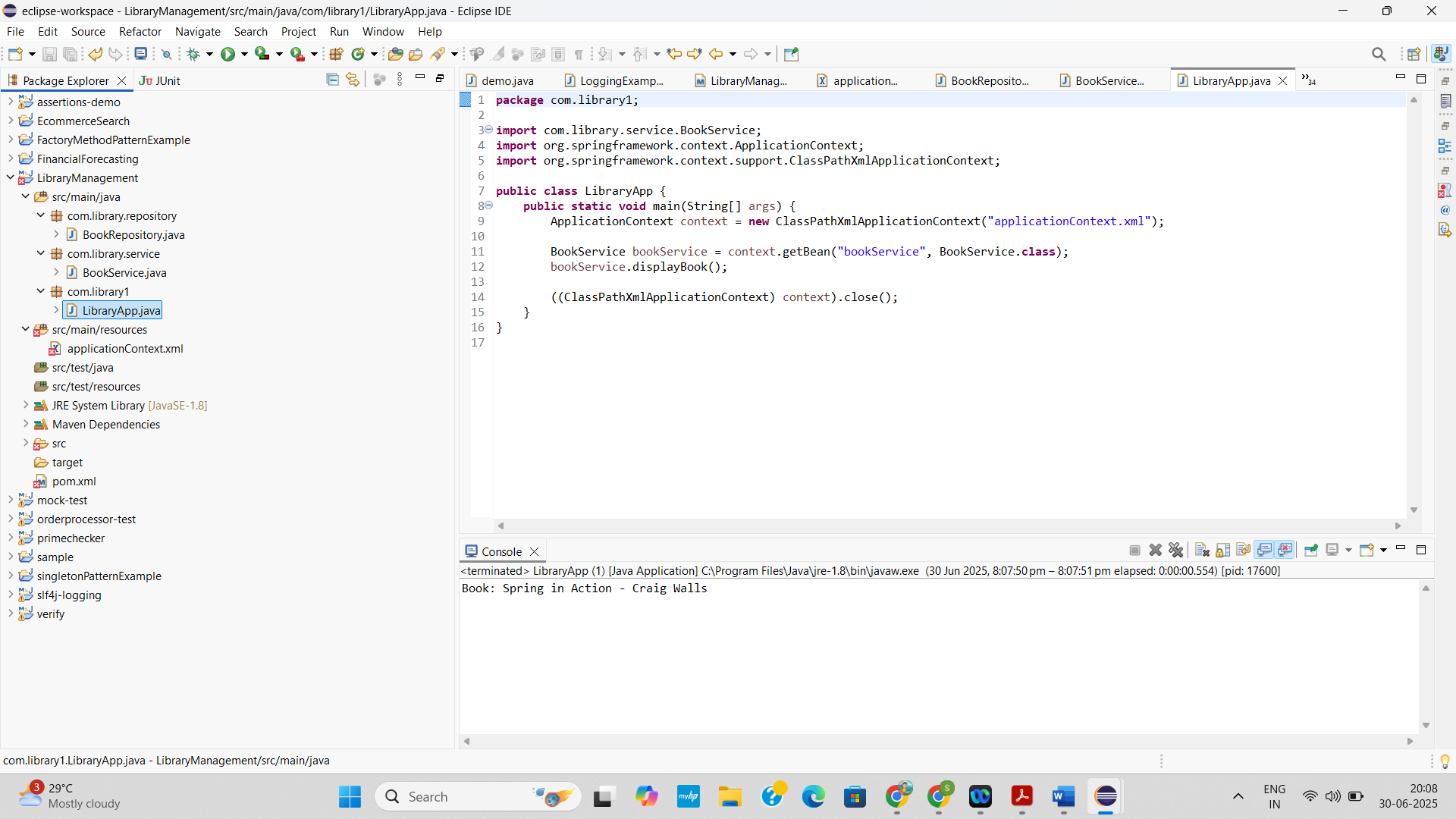
bookService.displayBook();

((ClassPathXmlApplicationContext) context).close();

}

}

**OUTPUT**

****

**EXERCISE 2: IMPLEMENTING DEPENDENCY INJECTION**

**SCENARIO**

In the library management application, you need to manage the dependencies between the BookService and BookRepository classes using Spring's IoC and DI.

**APPROACH**

* Define Beans: Declare bookRepository and bookService in applicationContext.xml.
* Setter Injection: Add a setter in BookService to receive the BookRepository.
* Bean Wiring: Link both beans using <property> in XML.
* Load Context: Use ClassPathXmlApplicationContext in your main class.
* Run & Test: Invoke service methods to confirm DI is working (check console output).

**CODE**

**pom.xml**

**Added following dependencies in pom.xml**

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.30</version> <!-- You can use latest 5.x version -->

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>5.3.30</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.36</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.7.36</version>

</dependency>

</dependencies>

**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 title) {

System.out.println("Book saved: " + title);

}}

**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 title) {

bookRepository.saveBook(title);

}

}

**LibraryManagementApplication.java**

package com.library1;

import com.library.service.BookService;

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");

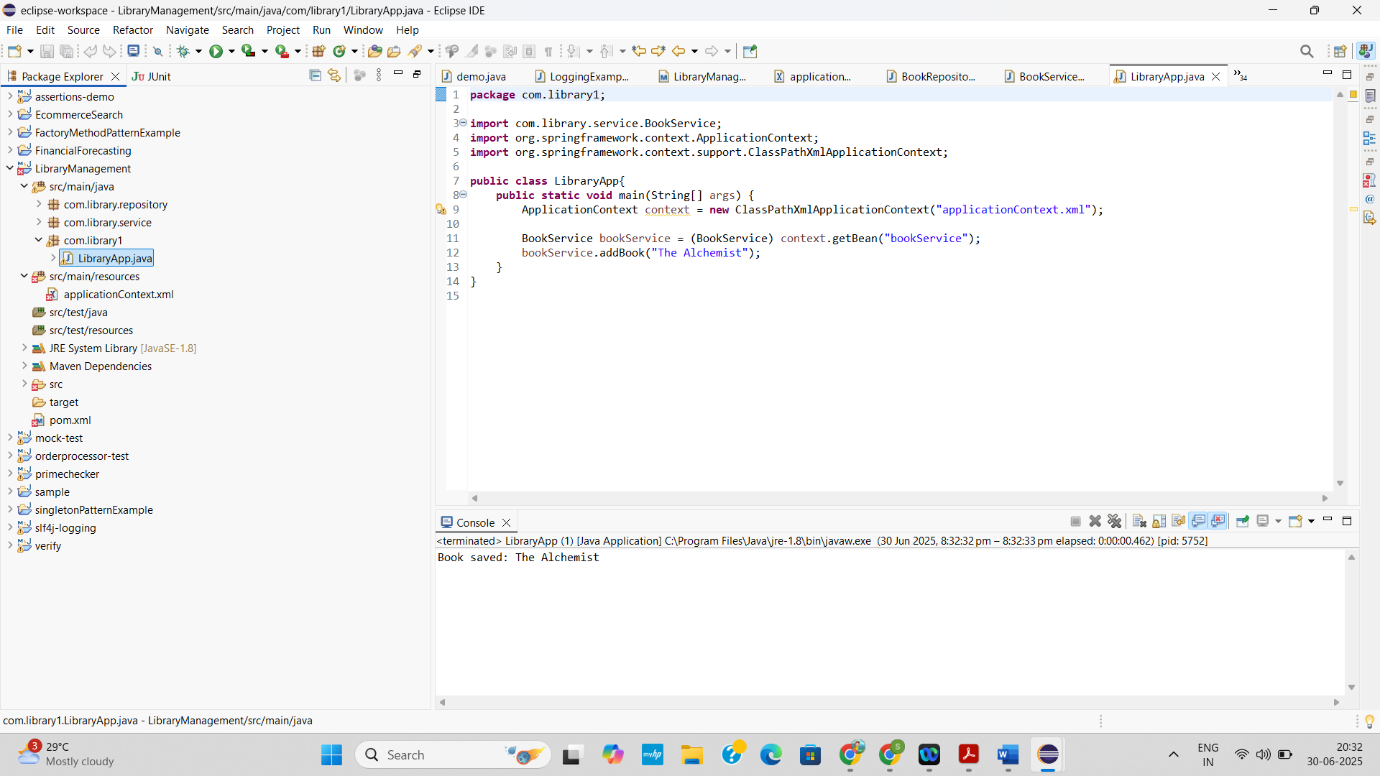
BookService bookService = (BookService) context.getBean("bookService");

bookService.addBook("The Alchemist");

}

}

**OUTPUT**

****

**EXERCISE 4: CREATING AND CONFIGURING A MAVEN PROJECT**

**SCENARIO**

You need to set up a new Maven project for the library management application and add Spring dependencies.

**APPROACH**

* Create a Maven project and add Spring dependencies in pom.xml.
* Write a simple service class (e.g., BookService) with a display method.
* Create applicationContext.xml and define the bean for BookService.
* In the main class, load the Spring context using ClassPathXmlApplicationContext.
* Retrieve the bean and call its method to verify Spring setup via console output.

**CODE**

**pom.xml**

**Added following dependencies in pom.xml**

<dependencies>

<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>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.11.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

**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="bookService" class="com.library.BookService"/>

</beans>

**BookService.java**

package com.library;

public class BookService {

public void display() {

System.out.println("📚 BookService is working via Spring!");

}

}

**App.java**

package com.library;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

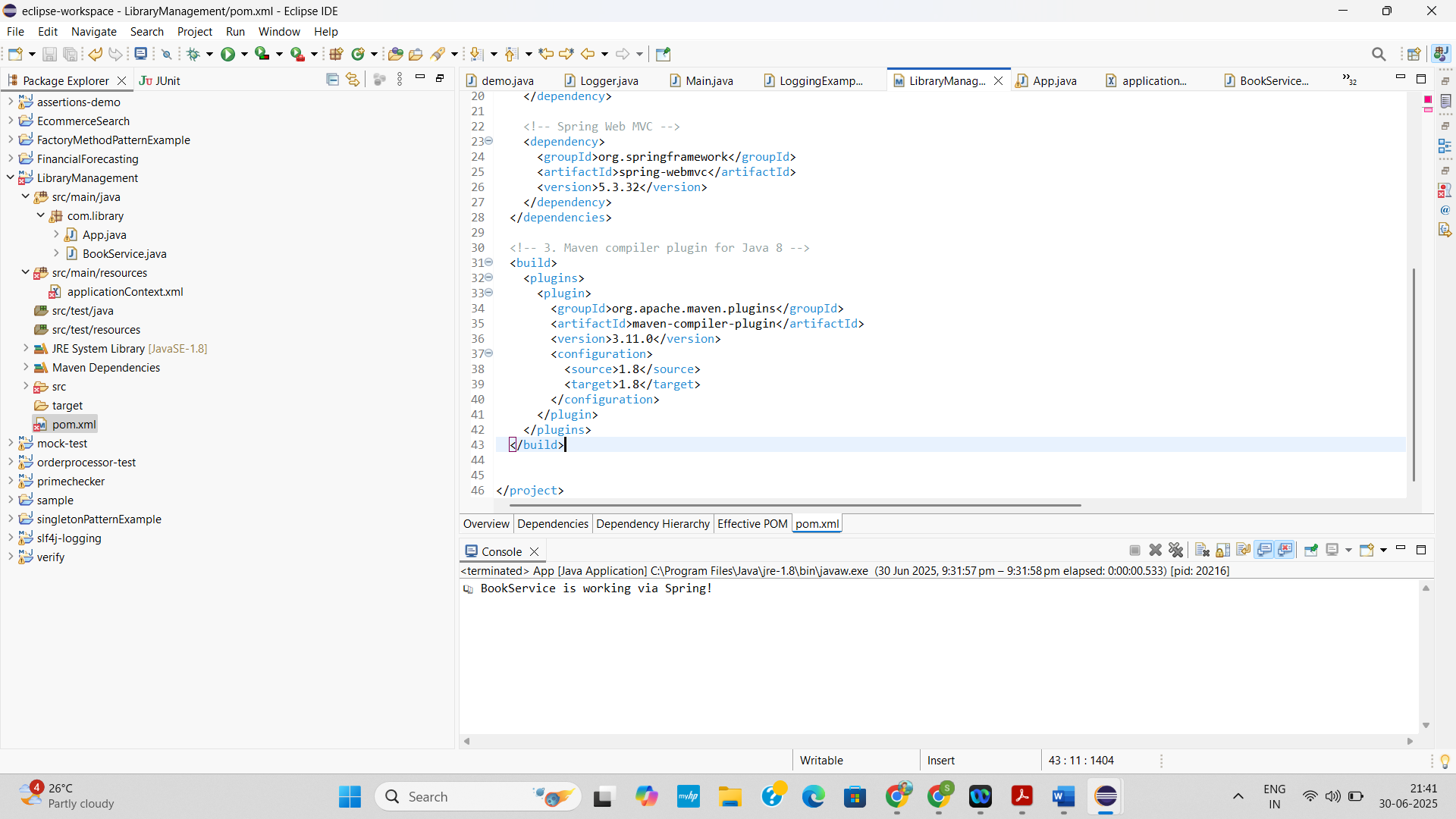
BookService service = context.getBean("bookService", BookService.class);

service.display();

}

}

**OUTPUT**

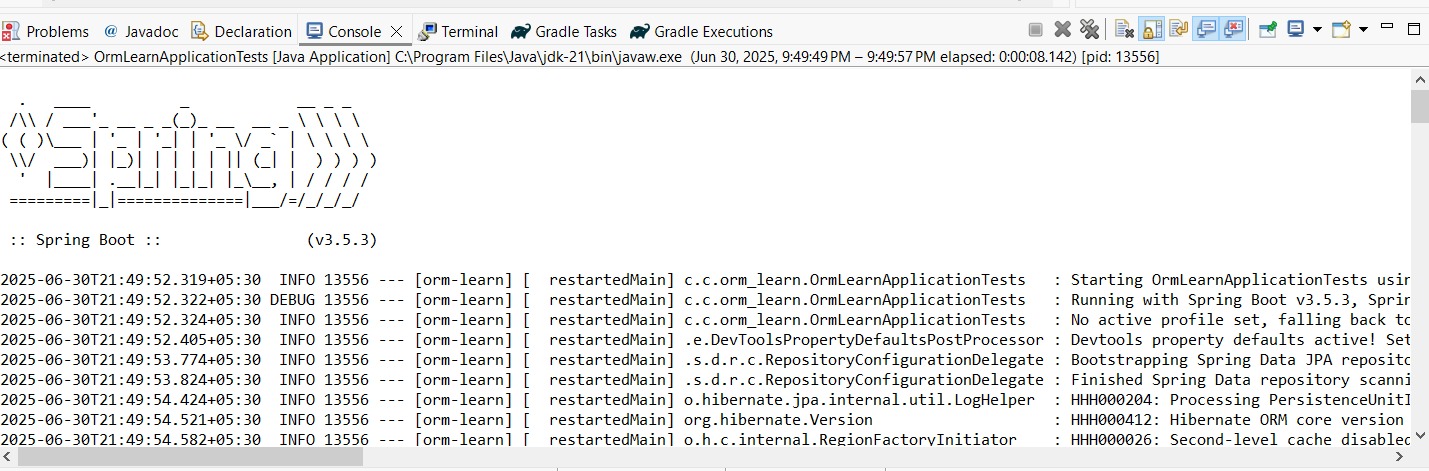


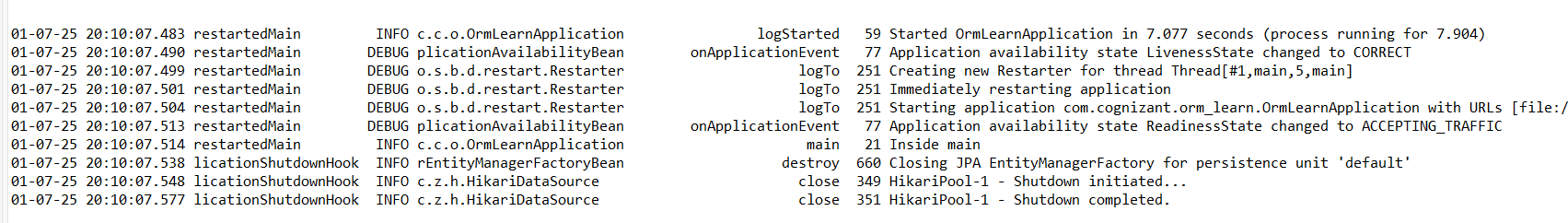
**SPRING DATA JPA WITH SPRING BOOT, HIBERNATE**

**EXERCISE 01: SPRING DATA JPA - QUICK EXAMPLE**

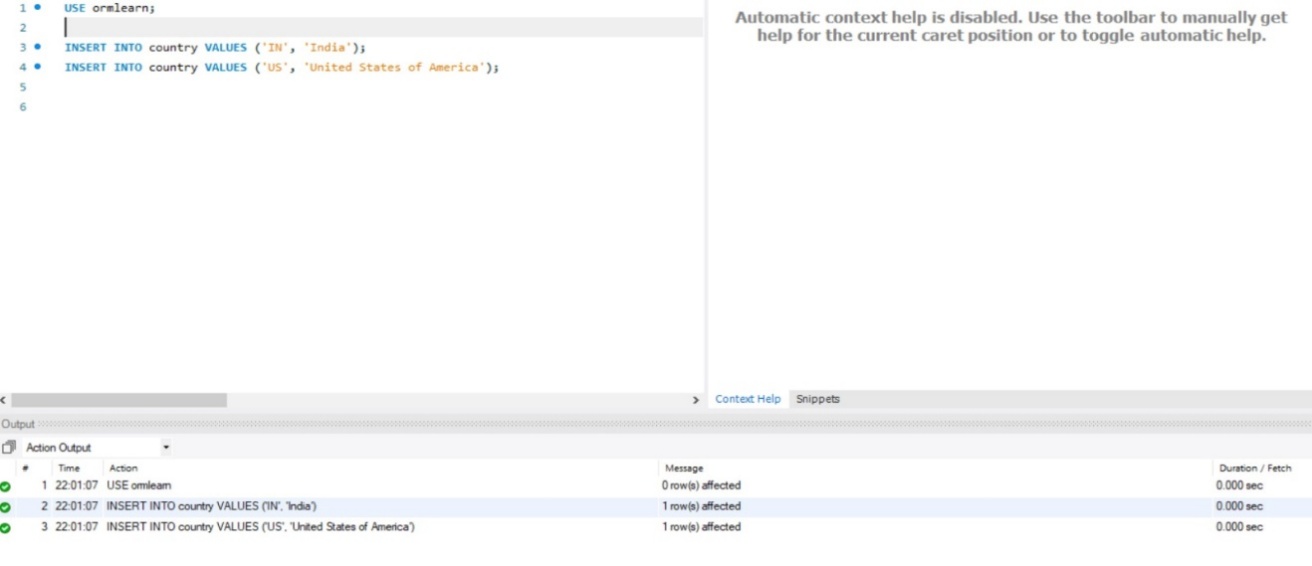
**APPROACH**

* Added the databases and log configuration in the application.properties file as per given.
* Included logs in the application and verified that the main() method is called. And the output is as follows:



………  


* And I also created table named “country” and entered the couple of records into the table as per the table.



* The “Country.java” class is an entity that represents the country table in the database, with fields code and name mapped using JPA annotations.
* The “CountryRepository.java” interface extends JpaRepository to provide built-in CRUD operations for the Country entity without the need for boilerplate code.
* The “CountryService.java” class serves as the service layer, where business logic is implemented; it autowires the CountryRepository and provides methods like getAllCountries() to retrieve data from the database.

**CODE**

**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 = "code")

private String code;

@Column(name = "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.model.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.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.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.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);

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

*testGetAllCountries*();

}

private static void testGetAllCountries() {

*LOGGER*.info("Start");

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

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

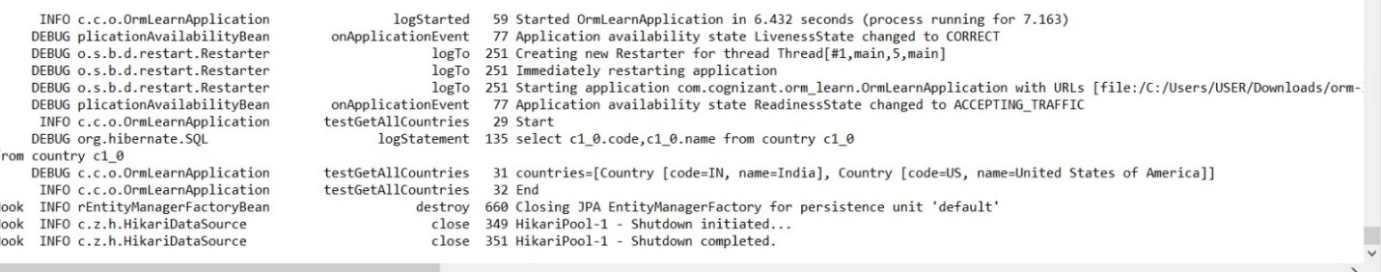
*LOGGER*.info("End");

}

}

**OUTPUT**





**EXERCISE 3: DIFFERENCE BETWEEN JPA, HIBERNATE AND SPRING DATA JPA**

**APPROACH**

**JPA (Java Persistence API):**

* + - A specification (like a rulebook) for how to manage relational data in Java using objects.
    - Part of Java EE (now Jakarta EE), defined under JSR 338.
    - It defines interfaces and annotations but has no implementation.

**Hibernate:**

* + - A popular ORM (Object-Relational Mapping) tool.
    - A concrete implementation of JPA.
    - Handles:
    - Connecting Java objects to DB tables
    - Generating SQL
    - Caching and lazy loading
    - Transaction management

**Spring Data JPA:**

* + - * A Spring framework project that simplifies JPA-based data access.
      * It does not implement JPA, but sits on top of JPA providers like Hibernate.
      * Provides:
        + Auto CRUD operations (no need to write DAO classes)
        + Repository interfaces
        + Built-in pagination, sorting, query generation

**CODE**

**Hibernate (Approach)**

**Dependencies added in pom.xml**

<dependencies>

<dependency>

<groupId>org.hibernate.orm</groupId>

<artifactId>hibernate-core</artifactId>

<version>6.4.4.Final</version>

</dependency>

<dependency>

<groupId>com.mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

<version>8.0.33</version>

</dependency

<dependency>

<groupId>jakarta.persistence</groupId>

<artifactId>jakarta.persistence-api</artifactId>

<version>3.1.0</version>

</dependency>

<dependency>

<groupId>org.jboss.logging</groupId>

<artifactId>jboss-logging</artifactId>

<version>3.5.3.Final</version>

</dependency>

</dependencies>  
**hibernate.cfg.xml (File created in src/main/resources)**

<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/testdb</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">Divya@priya135</property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>

<property name="hibernate.hbm2ddl.auto">update</property>

<property name="hibernate.show\_sql">true</property>

<mapping class="com.example.Employee"/>

</session-factory>

</hibernate-configuration>

**Employee.java**

package com.example;

import jakarta.persistence.\*;

@Entity

@Table(name = "employee")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

private int id;

private String name;

private String department;

// Getters & Setters

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getDepartment() { return department; }

public void setDepartment(String department) { this.department = department; }

}

**Main.java:**

package com.example;

import org.hibernate.Session;

import org.hibernate.Transaction;

public class Main {

public static void main(String[] args) {

Employee emp = new Employee();

emp.setName("Samyugtha");

emp.setDepartment("Sales Manager");

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction tx = null;

try {

tx = session.beginTransaction();

session.save(emp); // persist employee object

tx.commit();

} catch (Exception e) {

if (tx != null) tx.rollback();

e.printStackTrace();

} finally {

session.close();

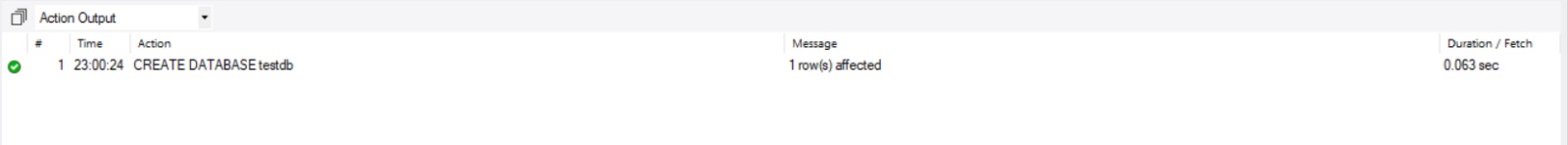
}

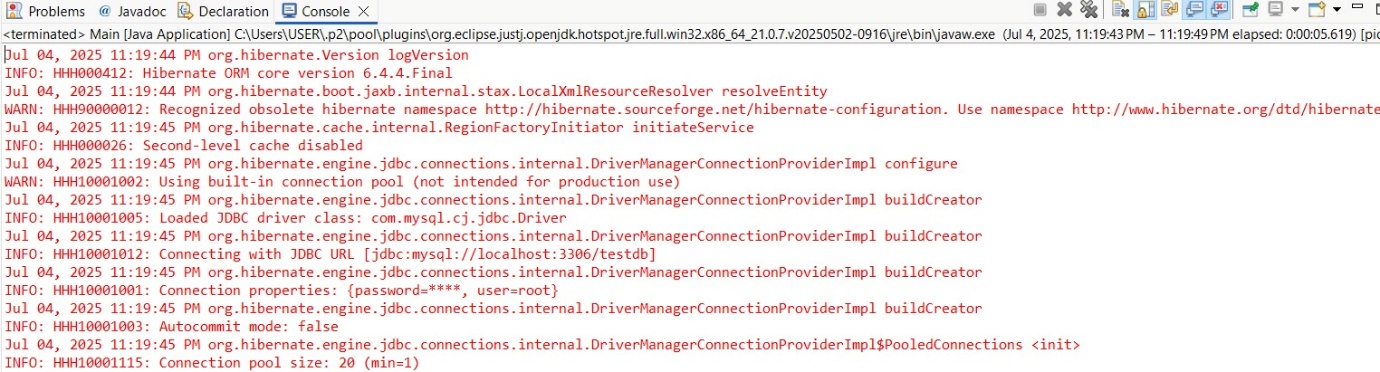
}

}

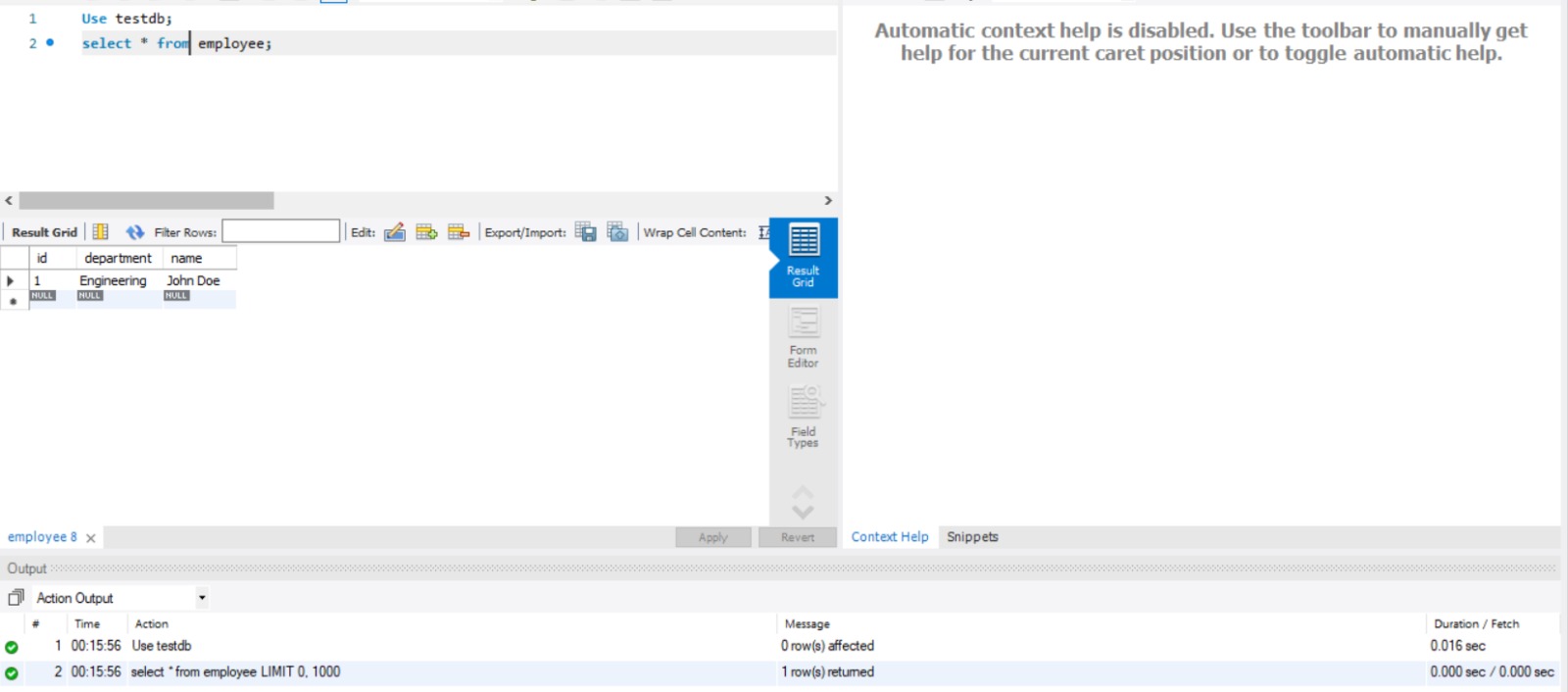
**OUTPUT**

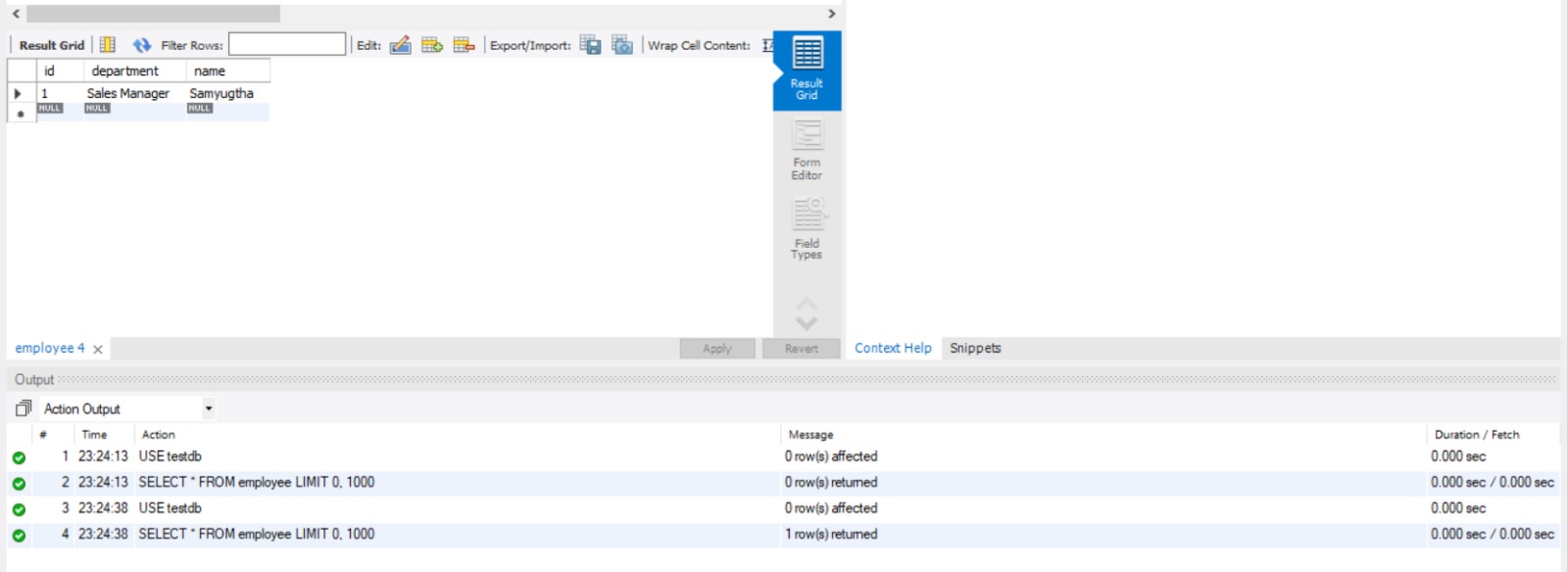
**Successfully created database testbase**



****

**Verified in MYSQL workbench**



****

**Explanation**

* Hibernate config (hibernate.cfg.xml) is loaded.
* SessionFactory(Database Factory) is created once, used throughout the app.
* A new Session(Database session) is opened to talk to the DB.
* A Transaction is started and committed to save data.
* On exception, transaction is rolled back**.**

**Spring Data JPA (Approach)**

**Dependencies added in pom.xml**

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>com.mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

<version>8.0.33</version>

</dependency>

<dependency>

<groupId>jakarta.persistence</groupId>

<artifactId>jakarta.persistence-api</artifactId>

<version>3.1.0</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>com.mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

<version>8.0.33</version>

</dependency>

</dependencies>

**application.properties (File created in src/main/resources)**

# Database connection

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

spring.datasource.username=root

spring.datasource.password=\*\*\*\*\*\*\* # “ \*\*\*\*\*\*\*\*” is the password for mysql

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

logging.level.org.hibernate.SQL=DEBUG

**Employee.java**

package com.example.entity;

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

private Integer id;

private String name;

private String department;

public void setName(String name) {

this.name = name;

}

public void setDepartment(String department) {

this.department = department;

}

// Optional: Add getters as well

public Integer getId() {

return id;

}

public String getName() {

return name;

}

public String getDepartment() {

return department;

}

}

**EmployeeRepository.java**

package com.example.repository;

import com.example.entity.Employee;

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

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

**EmployeeService.java**

package com.example.service;

import com.example.entity.Employee;

import com.example.repository.EmployeeRepository;

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

import org.springframework.stereotype.Service;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository repository;

public void saveEmployee(Employee employee) {

repository.save(employee);

System.*out*.println("Employee saved!");

}

}

**SpringDataJpaDemoApplication.java**

package com.example;

import com.example.entity.Employee;

import com.example.repository.EmployeeRepository;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringDataJpaDemoApplication implements CommandLineRunner {

@Autowired

private EmployeeRepository employeeRepository;

public static void main(String[] args) {

SpringApplication.*run*(SpringDataJpaDemoApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

Employee emp = new Employee();

emp.setName("Samyugtha");

emp.setDepartment("Archaeology");

employeeRepository.save(emp);

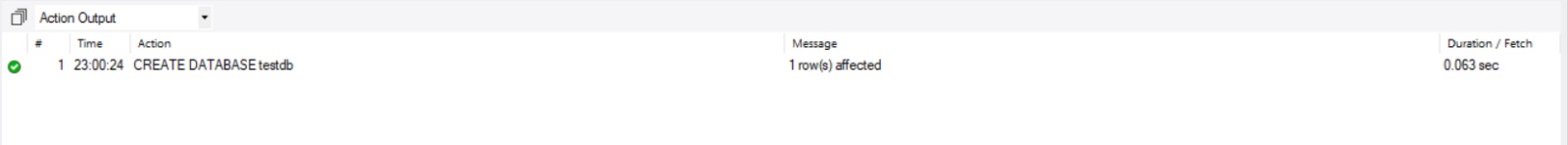
System.*out*.println("Employee Saved!");

}

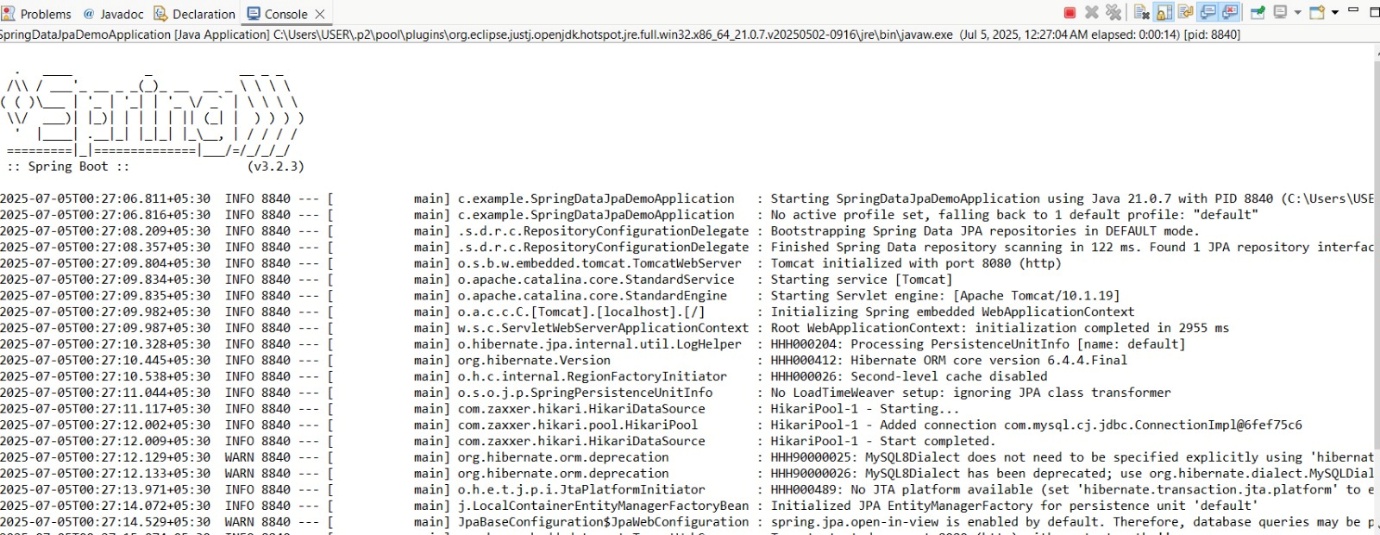
}

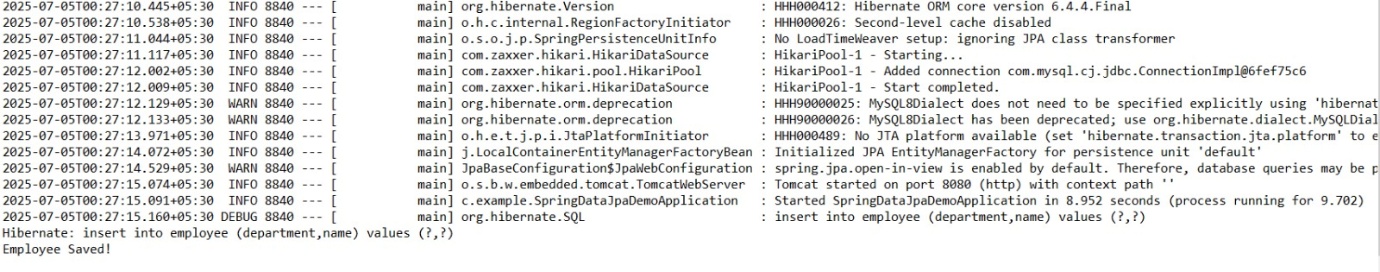
**OUTPUT**

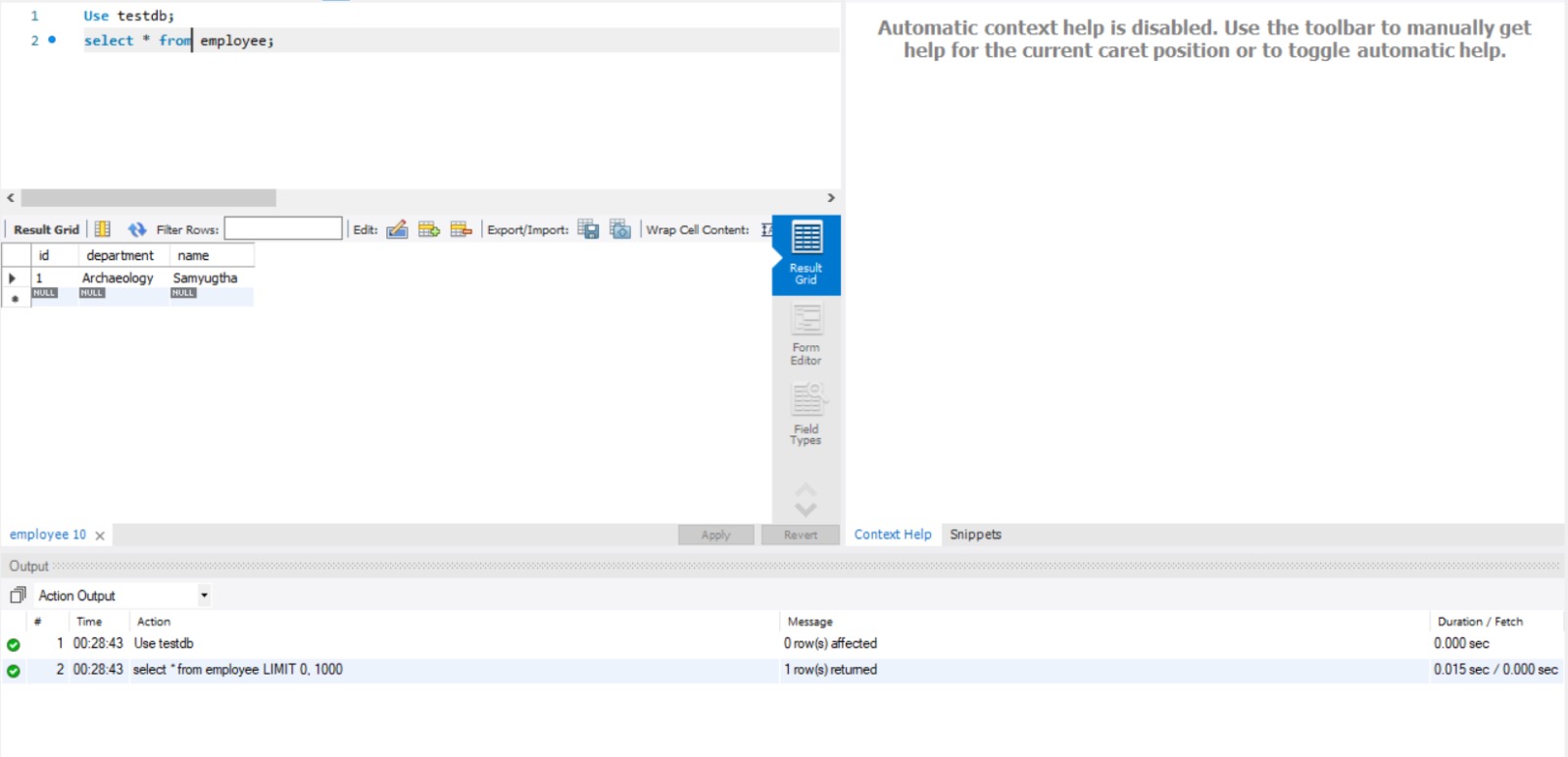
**Created database testdb**



**Console output after running SpringDataJpaDemoApplication as Java Application**

****

****

**Verified in MYSQL workbench** ****

**Explanation**

* Spring Data JPA simplifies data access in Java by providing an abstraction over JPA and Hibernate.
* Unlike traditional Hibernate code, it eliminates boilerplate by using built-in repository interfaces for CRUD operations.
* Implemented a basic Spring Data JPA project using Eclipse and Maven, configured the application with application.properties, created entity and repository classes,
* And it also verified successful data insertion into MySQL.