**Module 5 - 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.**

**Steps and Implementation:**

**1. Set Up a Spring Project (pom.xml)**

A Maven-based project named LibraryManagement was created using IntelliJ IDEA. The following dependency was added in the pom.xml to support Spring**:**

<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>

<dependencies>

<!-- Spring Context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.36</version>

</dependency>

</dependencies> </project>

**2. Configure the Application Context**

A Spring configuration file named applicationContext.xml was created in the src/main/resources directory to define the service and repository beans:

<?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">

<!-- Repository Bean -->

<bean id="bookRepository" class="com.library.repository.BookRepository" />

<!-- Service Bean with dependency injection -->

<bean id="bookService" class="com.library.service.BookService">

<property name="bookRepository" ref="bookRepository" />

</bean>

</beans>

**3. Define Service and Repository Classes**

**BookRepository.java**

package com.library.repository;

public class BookRepository {

public void fetchBooks() {

System.out.println("Fetching books from 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 listBooks() {

System.out.println("BookService: listing books");

bookRepository.fetchBooks();

}

}

**4. Run the Application**

**MainApp.java**

package com.library;

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) {

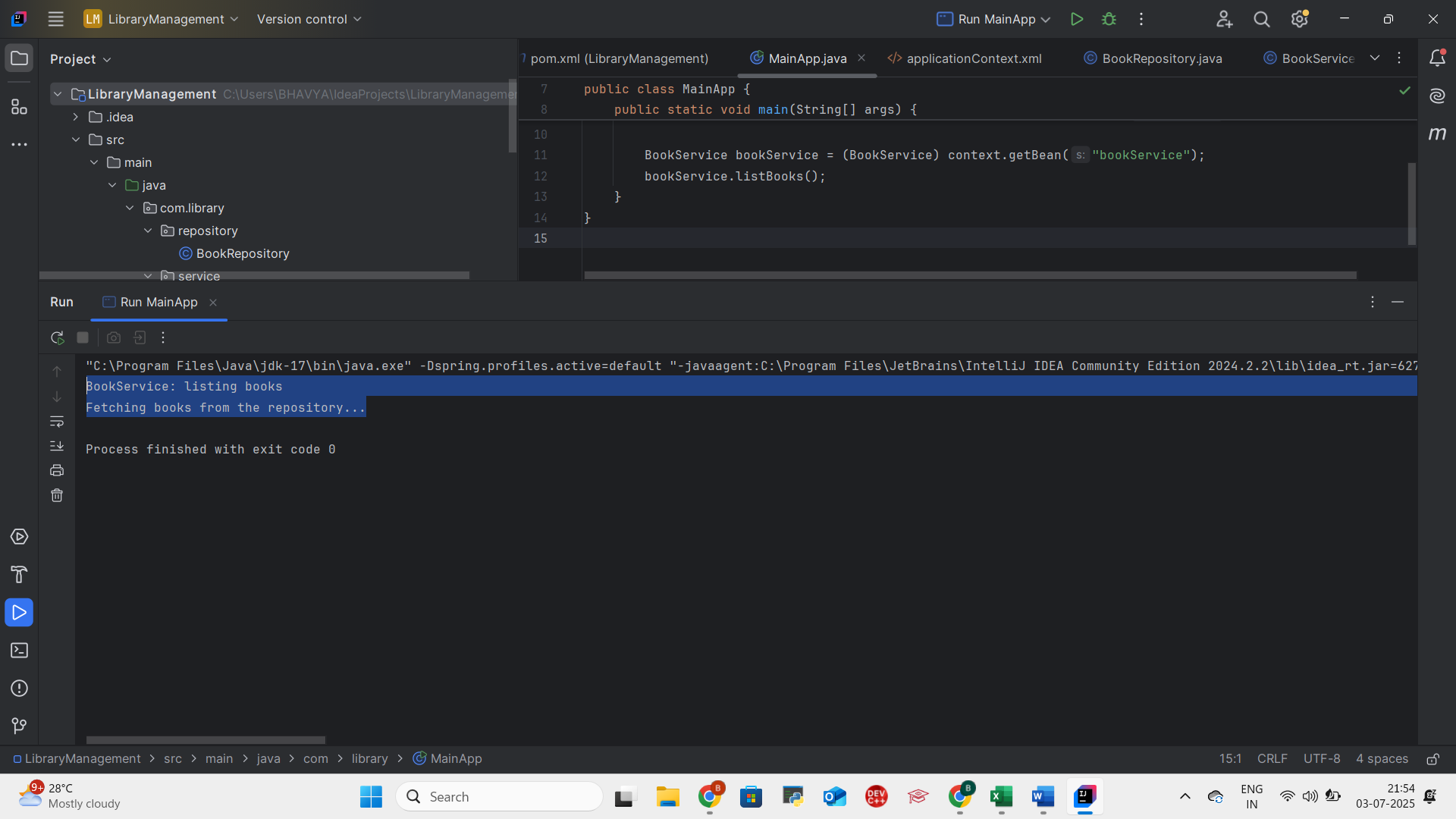
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

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

bookService.listBooks();

} }

**OUTPUT SCREENSHOT:**

****

**Conclusion**

This exercise demonstrates how to configure a basic Spring Framework application using XML-based bean configuration. The BookService and BookRepository classes were registered as beans in the Spring context and wired using setter injection. The Spring container successfully instantiated and injected dependencies, enabling the application to perform its operation.

**Exercise 2: Implementing Dependency Injection**

**Scenario**

**In the Library Management application, we need to manage the dependency between the BookService and BookRepository classes using Spring Framework’s Inversion of Control (IoC) and Dependency Injection (DI).**

**Steps and Implementation**

1. **Modify the XML Configuration**

We updated the applicationContext.xml file to define both the BookRepository and BookService beans. The BookService bean is configured with a property that refers to the BookRepository bean using setter-based dependency injection.

**src/main/resources/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">

<!-- Repository Bean -->

<bean id="bookRepository" class="com.library.repository.BookRepository" />

<!-- Service Bean with Dependency Injection -->

<bean id="bookService" class="com.library.service.BookService">

<property name="bookRepository" ref="bookRepository" />

</bean>

</beans>

**2. Update the BookService Class**

The BookService class is written to support setter-based dependency injection by providing a setBookRepository() method that Spring will use to inject the BookRepository bean.

**src/main/java/com/library/service/BookService.java**

package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

private BookRepository bookRepository;

// Setter method for Dependency Injection

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public void listBooks() {

System.out.println("BookService: listing books");

bookRepository.fetchBooks();

}

}

**3. Run the Application**

The MainApp class loads the Spring context from applicationContext.xml, retrieves the BookService bean, and invokes the listBooks() method.

**src/main/java/com/library/MainApp.java**

package com.library;

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.listBooks();

}

}

**4. BookRepository Class (No Change)**

**src/main/java/com/library/repository/BookRepository.java**

package com.library.repository;

public class BookRepository {

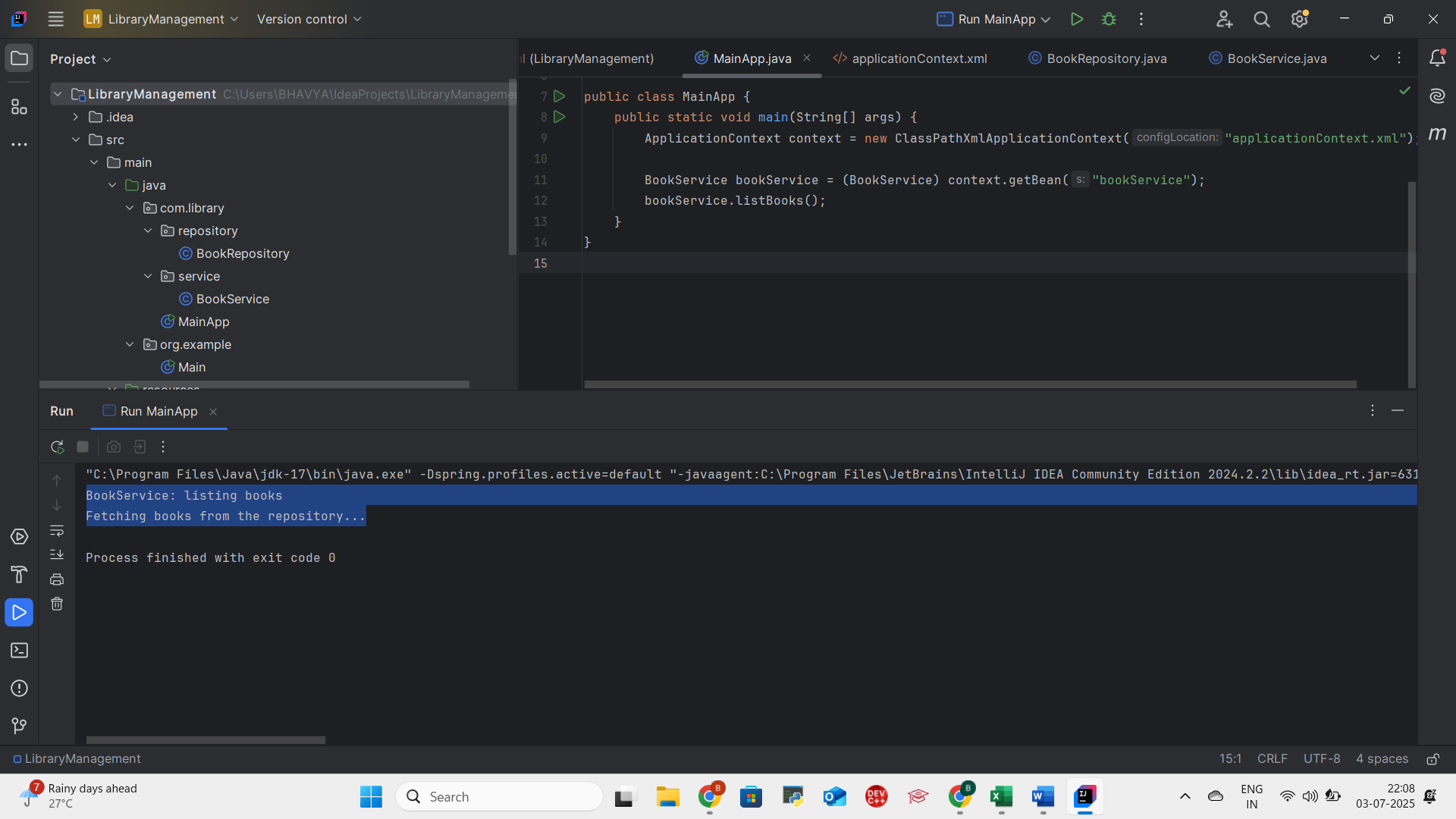
public void fetchBooks() {

System.out.println("Fetching books from the repository...");

}

}

**OUTPUT SCREENSHOT:**

****

**Conclusion**

In this exercise, Spring's Inversion of Control (IoC) container was used to implement Dependency Injection (DI) between BookService and BookRepository. Instead of manually instantiating and linking classes, Spring took control and injected dependencies using setter methods, as configured in applicationContext.xml.

**Although the output is the same as in Exercise 1, the key difference is the use of Spring’s DI pattern. This results in loosely coupled, easier-to-maintain, and testable code, which follows modern enterprise-level development practices.**

**Exercise 4: Creating and Configuring a Maven Project**

**Scenario**

**You need to set up a new Maven project for the Library Management application using IntelliJ IDEA and add essential Spring dependencies along with Maven plugins.**

**Steps and Implementation**

**1: Create a New Maven Project**

The Maven project is created using IntelliJ IDEA Community Edition:

1. Open IntelliJ
2. Click on File > New > Project
3. In the left panel, select Java
4. In the middle panel, under Build system, select Maven
5. Choose JDK: 17 Oracle OpenJDK 17.0.12
6. Uncheck Add sample code
7. Set the Name to LibraryManagement
8. Click Create

**2: Add Spring Dependencies in pom.xml**

<dependencies>

<!-- Spring Context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.36</version>

</dependency>

<!-- Spring AOP -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.36</version>

</dependency>

<!-- Spring WebMVC -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.36</version>

</dependency>

</dependencies>

**3: Configure Maven Compiler Plugin**

<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>

**4: Reload the Maven Project**

1. Open the **Maven** tab
2. Click the **Refresh icon**
3. IntelliJ downloads and syncs all dependencies from pom.xml

**Screenshot of pom.xml**

A screenshot of a computer program

AI-generated content may be incorrect.

**Conclusion**

In this exercise, a clean Maven project named **LibraryManagement** was created using IntelliJ IDEA. Spring dependencies for Context, AOP, and WebMVC were added successfully in the pom.xml file, and the Maven compiler plugin was configured to use Java 1.8. This setup ensures that the project is ready for further Spring-based development in upcoming exercises.