# **WEEK-03 HANDS ON SOLUTIONS**

# **SPRING CORE \_ MAVEN**

**EXERCISE-01 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:

1. Set Up a Spring Project:

o Create a Maven project named LibraryManagement.

o Add Spring Core dependencies in the pom.xml file.

2. Configure the Application Context:

o Create an XML configuration file named applicationContext.xml in the src/main/resources directory.

o Define beans for BookService and BookRepository in the XML file.

3. Define Service and Repository Classes:

o Create a package com.library.service and add a class BookService.

o Create a package com.library.repository and add a class BookRepository.

4. Run the Application:

o Create a main class to load the Spring context and test the configuration.

**SOLUTION:**

The objective of this exercise was to configure a basic Spring application for a library management system. The goal was to handle backend operations using the Spring Framework by setting up the service and repository layers and wiring them using Spring’s bean configuration mechanism.

A Maven project named LibraryManagement was created using Eclipse IDE. The Spring Core dependency (spring-context) was added to the pom.xml file to enable bean configuration and application context features.

An XML configuration file named applicationContext.xml was created inside the src/main/resources directory.

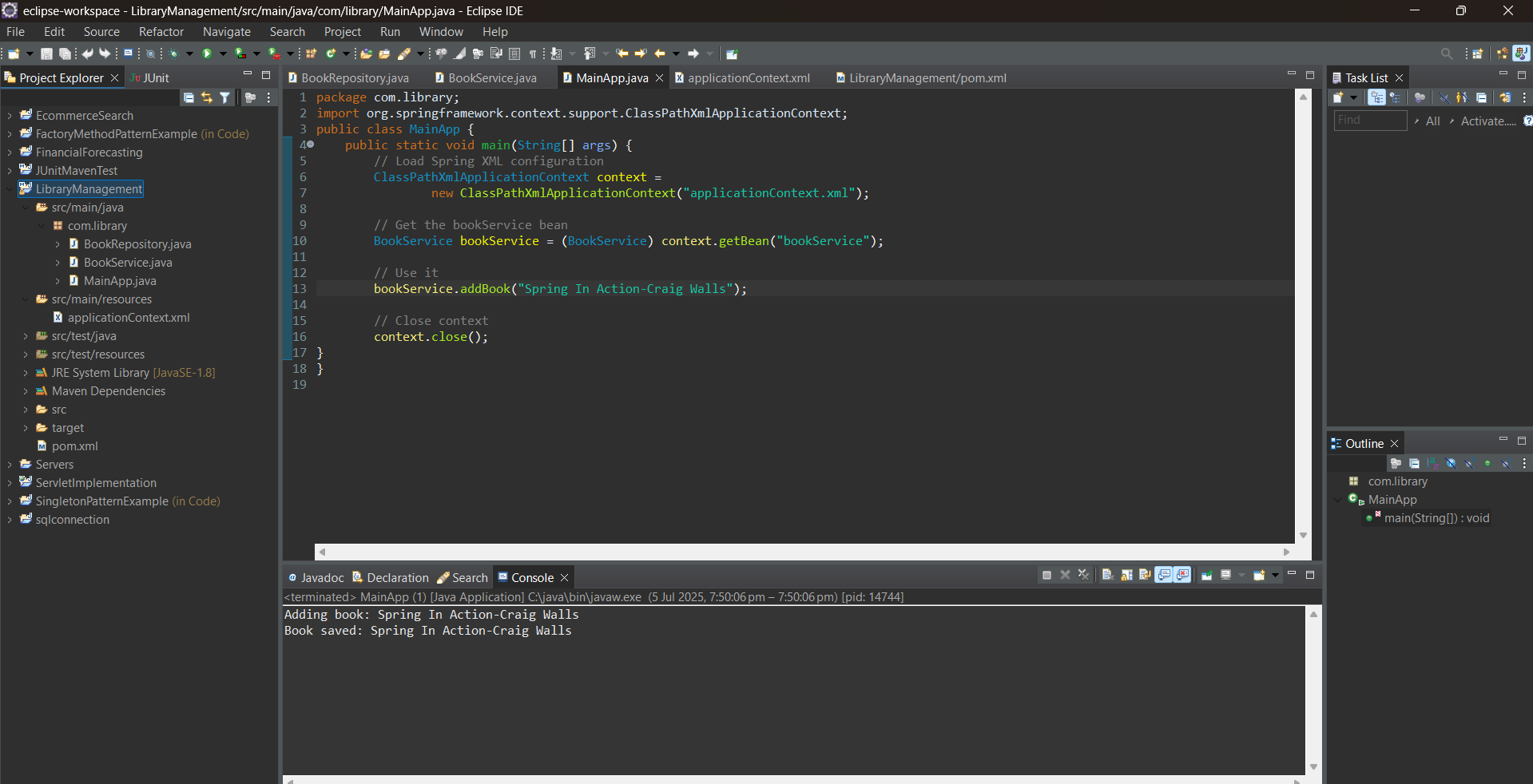
In this file, two beans were defined: bookRepository, representing the data access component bookService, representing the service layer that uses the repository The bookRepository bean was connected to the bookService using a <property> tag, allowing the service class to access repository methods.

The BookService class was placed under the com.library.service package and contained a method called addBook(String title).

The BookRepository class was created under the com.library.repository package and had a method saveBook(String title) that printed the saved book title to the console.

To test the setup, a MainApp class was created in the com.library package. It loaded the Spring configuration using ClassPathXmlApplicationContext, retrieved the bookService bean, and invoked the addBook() method with a sample book title.

**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.

Steps:

1. Modify the XML Configuration:

o Update applicationContext.xml to wire BookRepository into BookService.

1. Update the BookService Class:

o Ensure that BookService class has a setter method for BookRepository.

1. Test the Configuration:

o Run the LibraryManagementApplication main class to verify the dependency injection.

**SOLUTION:**

The aim of this exercise was to implement Dependency Injection (DI) in the library management application using Spring Framework.

DI is a core feature of Spring and is part of a broader concept known as Inversion of Control (IoC).

It allows the control of object creation and dependency management to be shifted from the application code to the Spring container.

This promotes loose coupling between classes, making the application more modular and testable.

To implement DI, the existing applicationContext.xml file was updated to explicitly wire the dependency between BookService and BookRepository.

In this setup, BookRepository acts as a dependency that needs to be provided to BookService.

Spring handles this by injecting the bookRepository bean into the bookService bean using XML configuration.

This was done through the <property> tag, which specifies the setter method to be used for injection.

At the Java class level, the BookService class already contained a setter method named setBookRepository(BookRepository bookRepository).

This method enabled Spring to inject an instance of BookRepository during the application context initialization.

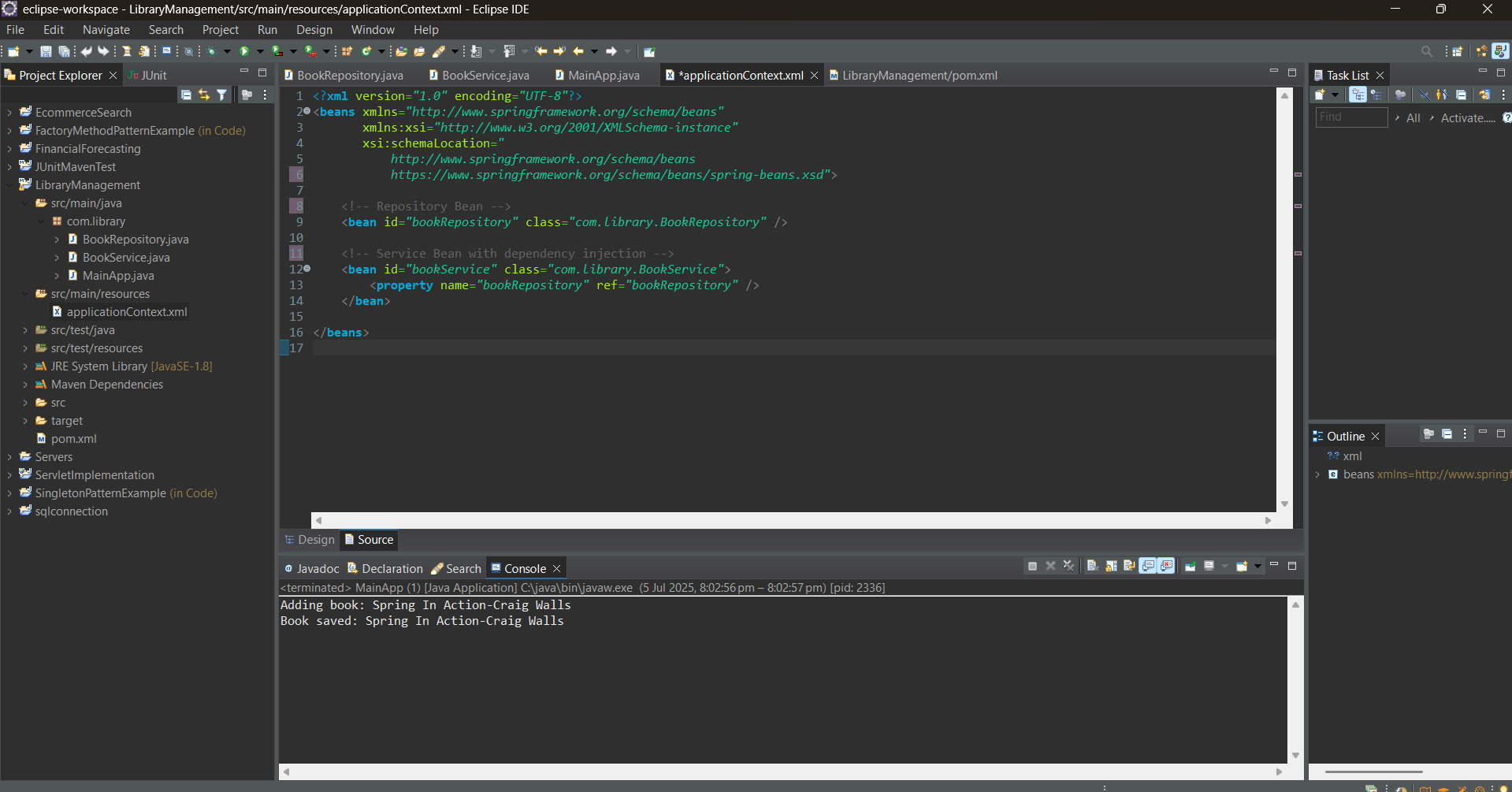
This approach follows setter-based injection, where dependencies are passed to the class after the object is constructed.

The injection was tested by running a main class (MainApp) that loaded the Spring context and retrieved the bookService bean.

The application then called the addBook() method, which internally invoked the saveBook() method from BookRepository.

This confirmed that the dependency was injected correctly and the beans were functioning together.

**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.

Steps:

1. Create a New Maven Project:

o Create a new Maven project named LibraryManagement.

2. Add Spring Dependencies in pom.xml:

o Include dependencies for Spring Context, Spring AOP, and Spring WebMVC.

3. Configure Maven Plugins:

o Configure the Maven Compiler Plugin for Java version 1.8 in the

pom.xml file.

**SOLUTION:**

The objective of this exercise was to create a new Maven-based project for the Library Management System and configure it with the necessary Spring dependencies and build plugins. This setup ensures that the project follows a structured, modular build process and is ready for Spring application development.

A new Maven project named LibraryManagement was created using Eclipse IDE.

Maven was selected as the build tool for its powerful dependency management and project structure features.

It simplifies the addition of external libraries such as Spring, and helps maintain consistency across environments.

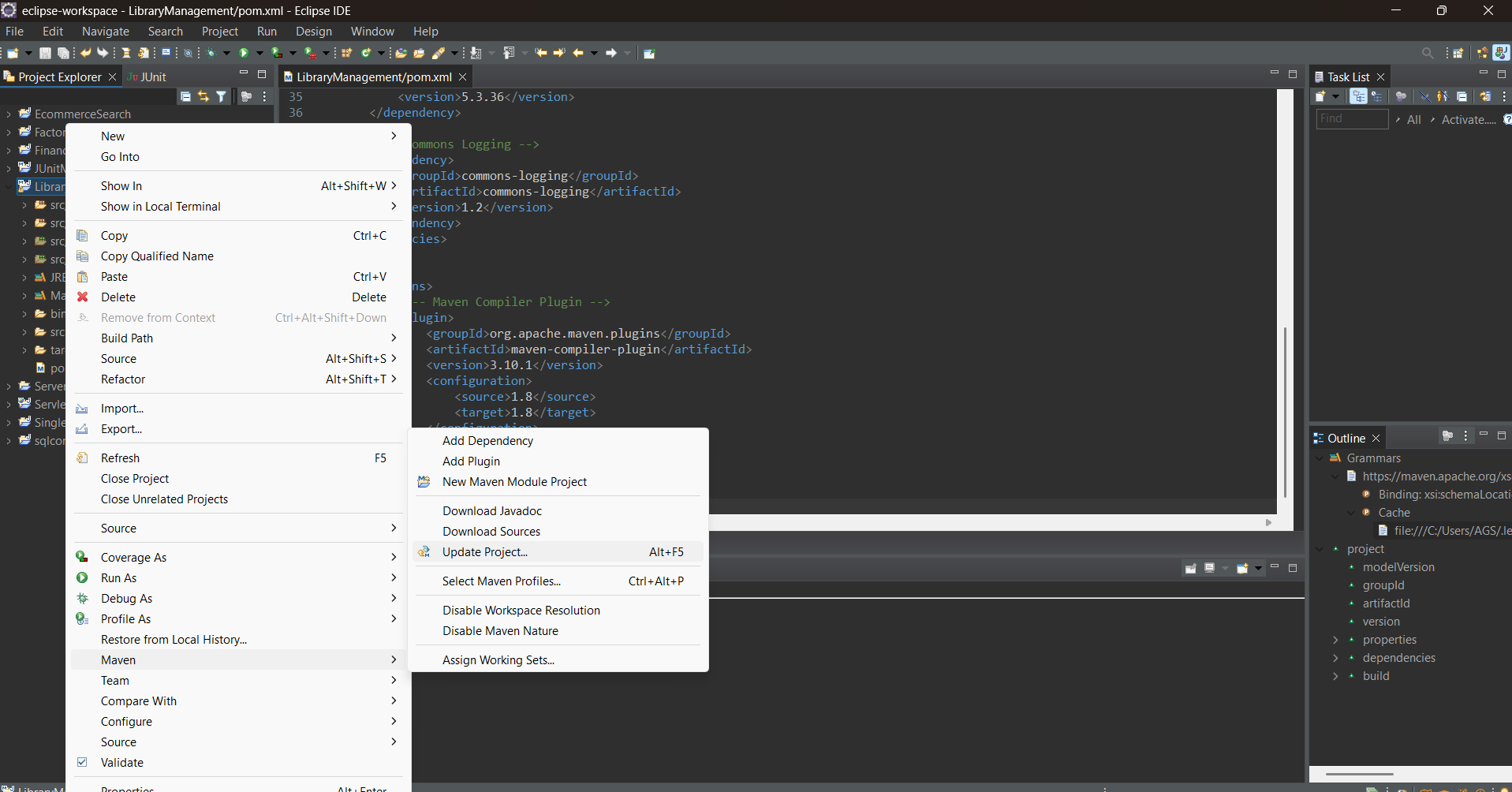
The next step involved editing the pom.xml file to add required Spring dependencies.

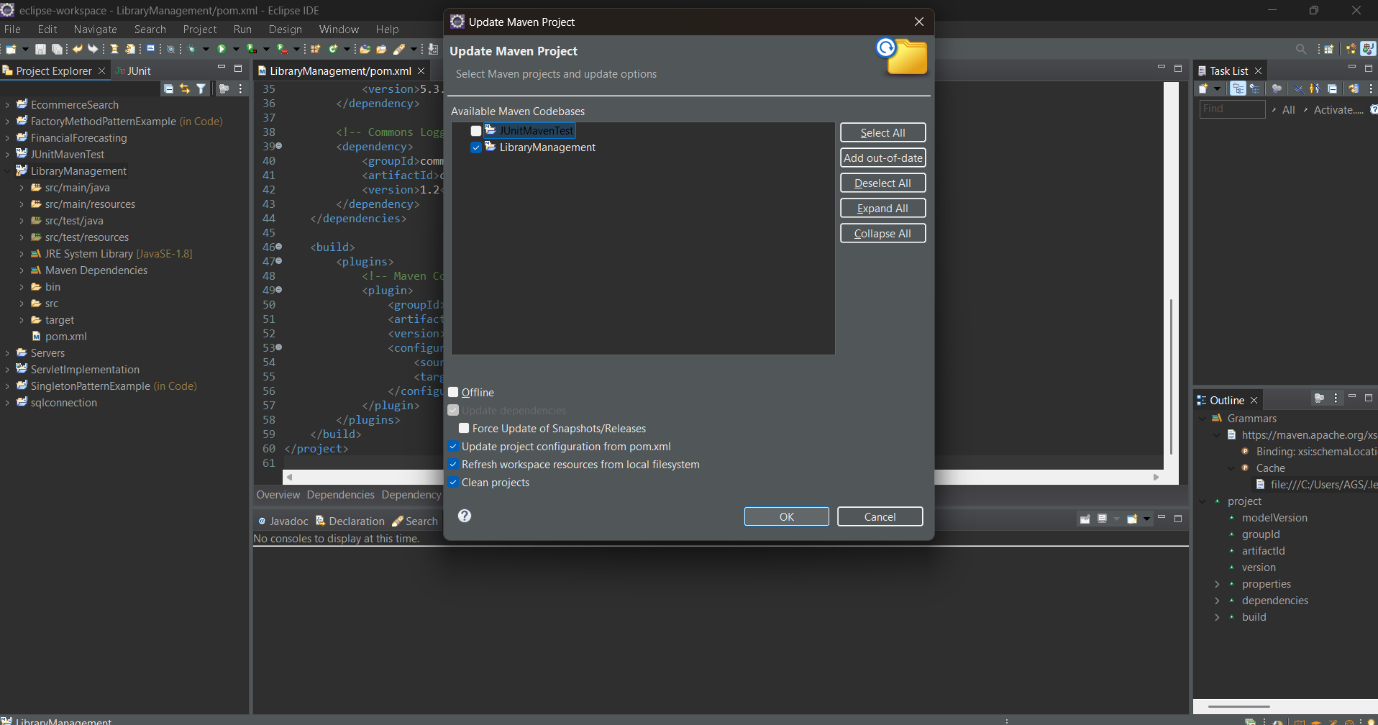
Specifically, three key Spring modules were included: Spring Context, for managing the application context and bean lifecycle Spring AOP, for supporting aspect-oriented programming (to be used in future exercises) Spring WebMVC, for handling web-related components like controllers and views.

After adding the dependencies, the Maven Compiler Plugin was configured in the pom.xml to ensure that the project uses Java version 1.8 for both source and target compilation. This step ensures compatibility with Spring 5.x and avoids runtime issues related to Java version mismatches.

Once the configuration was complete, Maven automatically downloaded the required Spring JAR files and indexed them in the project's classpath. No errors were reported, and the build process succeeded. This confirmed that the Maven setup was correctly configured and all dependencies were resolved.

**OUTPUT:**

****

****