**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.

**Step1 : Set Up a Spring Project**

**Create Maven Project**

Directory structure:

LibraryManagement/

.src/

. main/

. java/

. resources/

. pom.xml

**Add Spring Core Dependency in pom.xml:**

<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 Core -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.33</version> <!-- Or latest version -->

</dependency>

</dependencies>

</project>

**Step 2: Configure the Application Context**

**Create XML File: 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">

<!-- BookRepository Bean -->

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

<!-- BookService Bean, injecting BookRepository -->

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

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

</bean>

</beans>

**Step 3 : Define Service and Repository Classes**

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

package com.library.repository;

public class BookRepository {

public void fetchBooks() {

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

}

}

com/library/service/BookService.java

package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

private BookRepository bookRepository;

// Setter for dependency injection

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public void displayBooks() {

bookRepository.fetchBooks();

System.out.println("Displaying books...");

}

}

**Step 4: Run the Application**

**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) {

// Load Spring configuration

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

// Retrieve the bookService bean

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

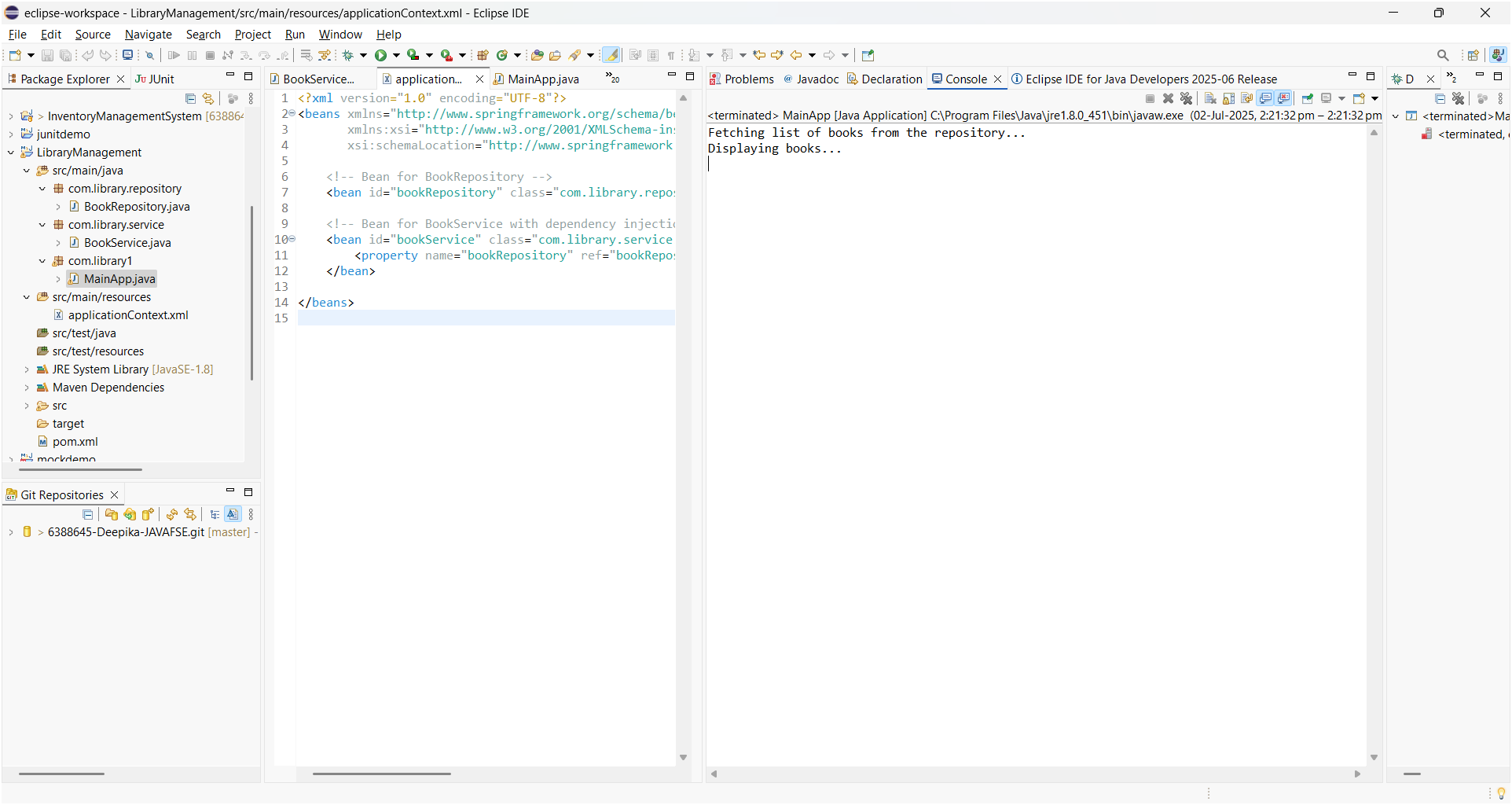
// Use the service

bookService.displayBooks();

}

}

**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.**

**Step 1: Modify the XML Configuration**

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

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

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

</bean>

**Step 2: Update the BookService Class**

To enable setter-based injection, a setter method in the BookService class that matches the property name in the XML configuration.

BookService.java:

package com.library.service;

import com.library.repository.BookRepository;

public class BookService

{

private BookRepository bookRepository;

// Setter for dependency injection

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public void displayBooks()

{

bookRepository.fetchBooks();

System.out.println("Books displayed.");

}

}

Step 3: Test the Configuration:

Created a main class to load the Spring container and test whether the dependency was successfully injected.

LibraryManagementApplication.java:

package com.library;

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");

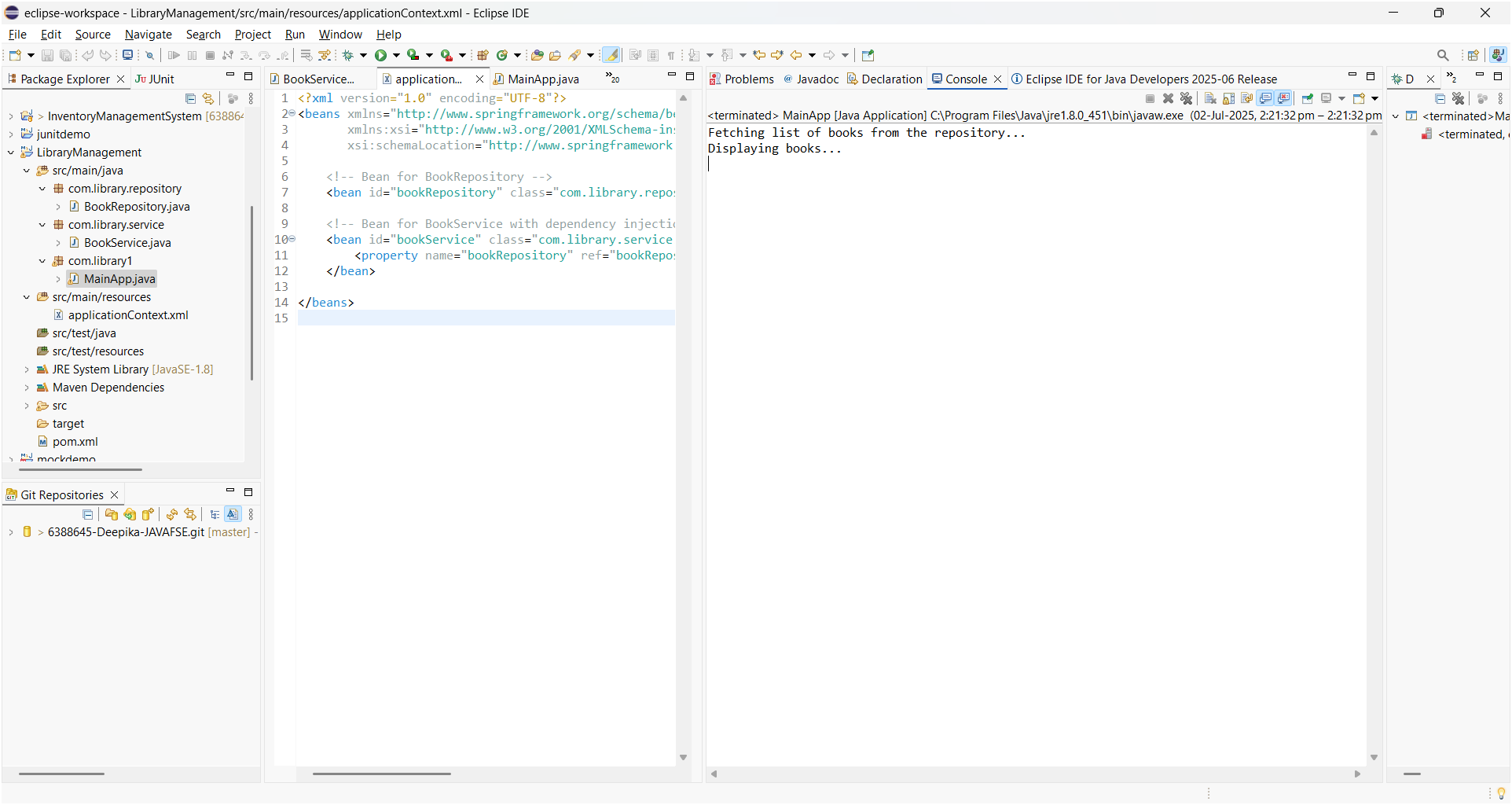
BookService bookService = (BookService) context.getBean("bookService");

bookService.displayBooks();

}

}

Successfully implemented **setter-based dependency injection** using Spring's IoC container. The BookRepository was injected into BookService via the XML configuration, and the setup was verified through output in the console.



Now observing the difference between

* Exercise 1: Configuring a Basic Spring Application
* Exercise 2: Implementing Dependency Injection

They are closely related, but their focus is different.

**Key Difference Between Exercise 1 and Exercise 2**

| Feature | Configuring Spring App | Implementing Dependency Injection |
| --- | --- | --- |
| Main Focus | Setting up the Spring project structure | Connecting components using Spring's DI mechanism |
| Objective | Create and configure the basic application | Inject BookRepository into BookService via Spring |
| applicationContext.xml | Create and register beans | Update beans to include <property> for injection |
| Testing | Run to verify Spring setup | Run to verify DI works and dependencies are injected |

In Simple Words,

Exercise 1 = Spring Setup

* A Maven project
* Add Spring Core dependency
* Build basic structure (BookService, BookRepository)
* Register beans in XML

Goal: Learn how to structure a Spring application and wire basic components.

Exercise 2 = Dependency Injection (DI)

* Modify the XML to add a <property> tag
* Add a setter in BookService
* Let Spring inject BookRepository into BookService

Goal: Understand how Spring manages object dependencies (IoC container).

**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.**

**Step1:**

Create a new Maven project named LibraryManagement using your IDE (like Eclipse/IntelliJ) or using the Maven command line.

**Step2:**

Inside the pom.xml file:

* Add dependencies for:
  + Spring Context (for core Spring features),
  + Spring AOP (for aspect-oriented programming),
  + Spring WebMVC (for web-related development).
* Configure the Maven Compiler Plugin to set both the source and target Java version to 1.8.

**Introducing Maven for dependency management:**

This task upgrades the project by introducing Maven for dependency management, making the setup more scalable and professional. It replaces the manual handling of library files by automatically managing all dependencies through the pom.xml.

Also, this version introduces support for additional Spring modules:

* Spring AOP allows handling cross-cutting concerns like logging or transactions.
* Spring WebMVC prepares the project for web controller development.

**Pom.xml:**

<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 Core (Context) -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.33</version>

</dependency>

<!-- Spring AOP -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.33</version>

</dependency>

<!-- Spring WebMVC -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.33</version>

</dependency>

</dependencies>

<build>

<plugins>

<!-- Maven Compiler Plugin -->

<plugin>

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

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

<version>3.8.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

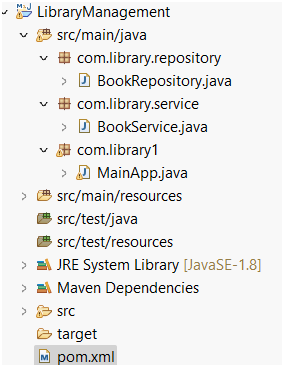
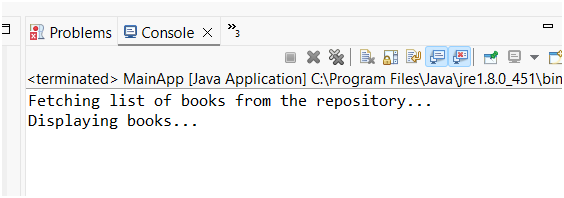
</plugin>

</plugins>

</build>

</project>

**Output:**



* The Maven project was successfully created and built without any errors.
* All required Spring dependencies (Context, AOP, WebMVC) were downloaded automatically and are visible under **Maven Dependencies** in the IDE.
* The project compiles successfully using Java 1.8, as verified by the Maven Compiler Plugin configuration.
* The pom.xml file reflects all necessary dependencies and plugin settings.