**WEEK-03-ADDITIONAL EXERCISES SOLUTIONS**

***-AAISHA SULTANA GUDURU-6384619***

**Exercise 5: Configuring the Spring IoC Container**

Scenario:

The library management application requires a central configuration for beans and dependencies.

Steps:

1. Create Spring Configuration File:

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.

2. Update the BookService Class:

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

3. Run the Application:

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

IMPLEMENTATION:

The objective of this exercise was to demonstrate the configuration and usage of the Spring Inversion of Control (IoC) container using XML-based configuration to manage dependencies between components in a library management system. Specifically, the goal was to configure the BookService and BookRepository beans, enable setter-based dependency injection, and verify the configuration through a simple Java application.

**1. Spring XML Configuration (applicationContext.xml)**

An XML configuration file named applicationContext.xml was created under the src/main/resources directory. This file defines the beans and their dependencies:

* **Bean Definition for BookRepository**: A bean with id bookRepository was declared, representing the data access component responsible for managing book data.
* **Bean Definition for BookService**: A bean with id bookService was defined. The BookService bean declares a property named bookRepository which is injected via setter injection referencing the bookRepository bean. This setup allows Spring IoC container to manage the dependency wiring.

The XML configuration enables loose coupling between service and repository layers and leverages Spring's container to handle object lifecycle and dependencies.

**2. Java Classes**

* **BookRepository Class**: A simple class representing the data repository. It includes an overridden toString() method for identification during output.
* **BookService Class**: Contains a private field for BookRepository and a public setter method setBookRepository() to support Spring’s setter-based injection. It also provides a method printBookDetails() to demonstrate usage of the injected dependency.

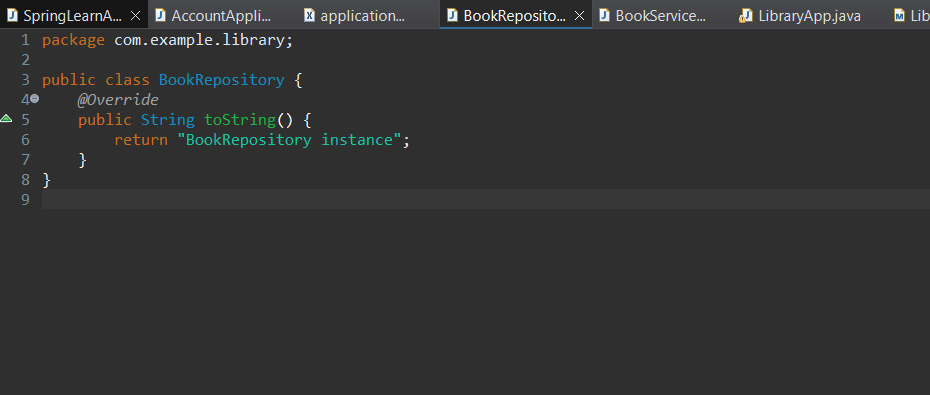
**3. Application Entry Point (LibraryApp)**

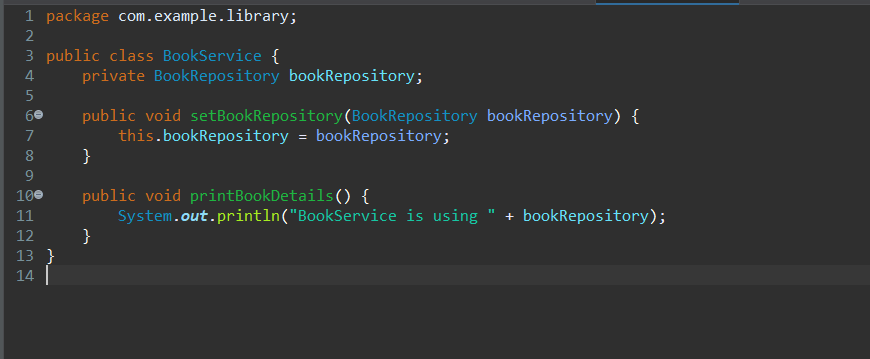
The main class LibraryApp loads the Spring IoC container by reading the applicationContext.xml from the classpath using ClassPathXmlApplicationContext. It retrieves the bookService bean from the context and invokes printBookDetails(), confirming the successful injection and configuration of dependencies.

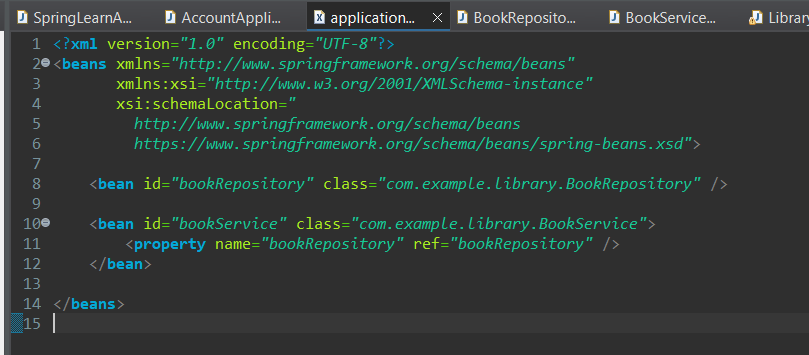
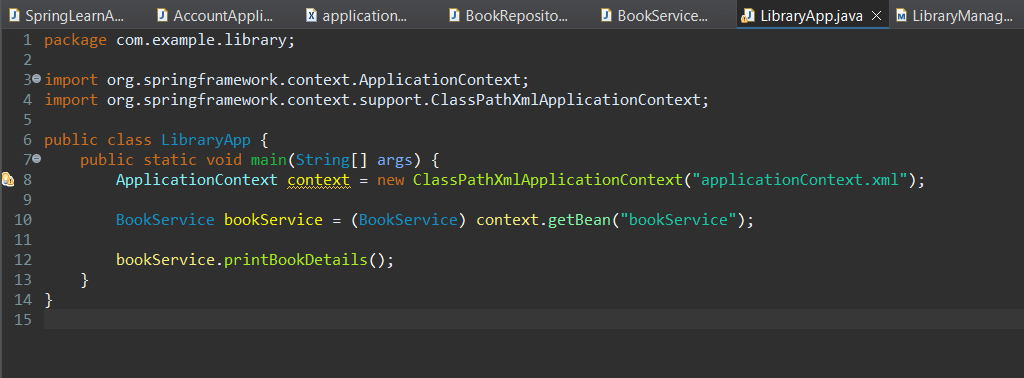
This exercise effectively demonstrated the fundamental concepts of Spring's IoC container, including:

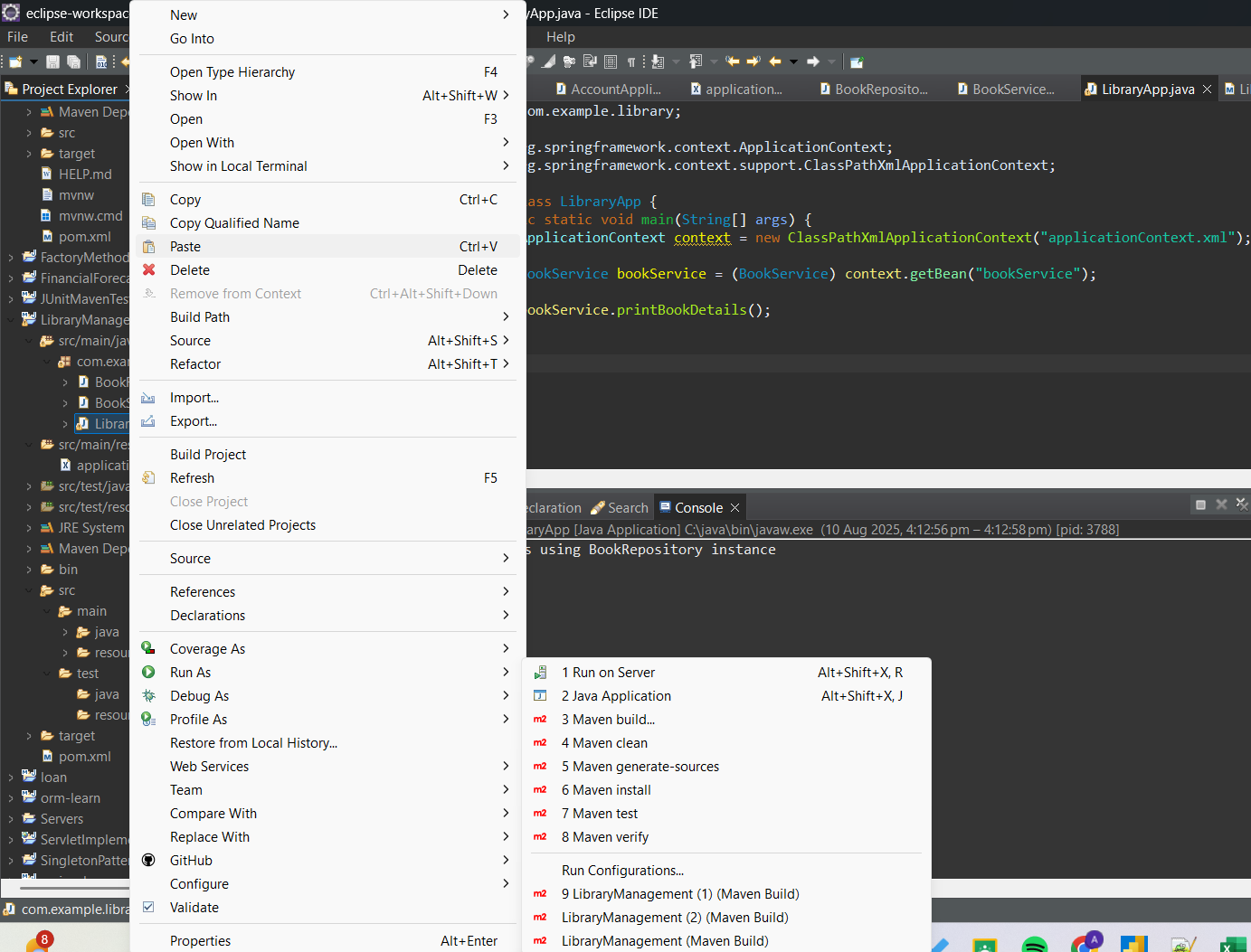
* Defining beans in an XML configuration file.
* Setting up setter-based dependency injection.
* Loading the Spring context and retrieving configured beans.
* Verifying bean wiring through a practical example.

This configuration pattern improves modularity and maintainability by decoupling component dependencies, allowing the Spring framework to manage object lifecycles and dependencies declaratively.

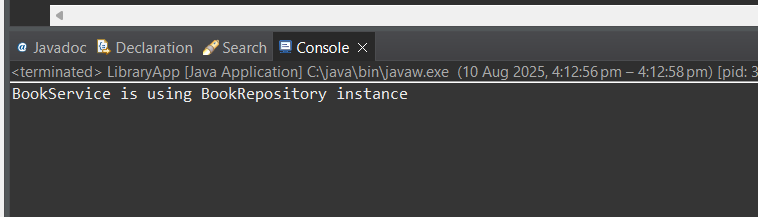








OUTPUT:



**Exercise 7: Implementing Constructor and Setter Injection**

Scenario:

The library management application requires both constructor and setter injection for better control over bean initialization.

Steps:

1. Configure Constructor Injection:

o Update applicationContext.xml to configure constructor injection for BookService.

2. Configure Setter Injection:

o Ensure that the BookService class has a setter method for BookRepository and configure it in applicationContext.xml.

3. Test the Injection:

o Run the LibraryManagementApplication main class to verify both constructor and setter injection.

**IMPLEMENTATION:**

The purpose of this exercise was to demonstrate the use of both constructor and setter-based dependency injection in a Spring IoC container. This approach provides better control over bean initialization and flexibility in configuring dependencies.

**1. Updating the BookService Class**

The BookService class was modified to support both constructor and setter injection for its dependency on BookRepository:

* A **constructor** was added that accepts a BookRepository instance, enabling constructor injection.
* A **setter method** setBookRepository() was implemented to support setter injection.

This dual approach allows Spring to inject dependencies either during object creation or after via setter, illustrating both methods in a single class.

**. Configuring applicationContext.xml**

The Spring XML configuration file applicationContext.xml was updated as follows:

* The BookRepository bean was declared to represent the repository component.
* The BookService bean was configured with both:
  + <constructor-arg> referencing the BookRepository bean to perform constructor injection.
  + <property> referencing the same BookRepository bean to perform setter injection.

This configuration ensures Spring injects the dependency using both methods, with setter injection overriding the constructor injection where applicable.

**3. Testing the Configuration**

The main application class LibraryApp loads the Spring context and retrieves the BookService bean. Invoking the printBookDetails() method confirms that the BookRepository dependency was successfully injected.

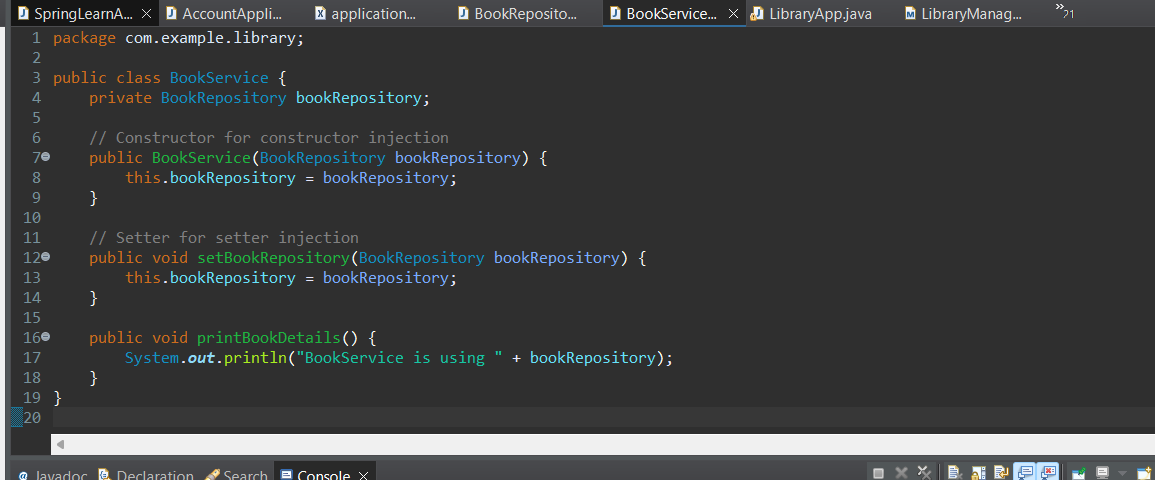
Output verified:

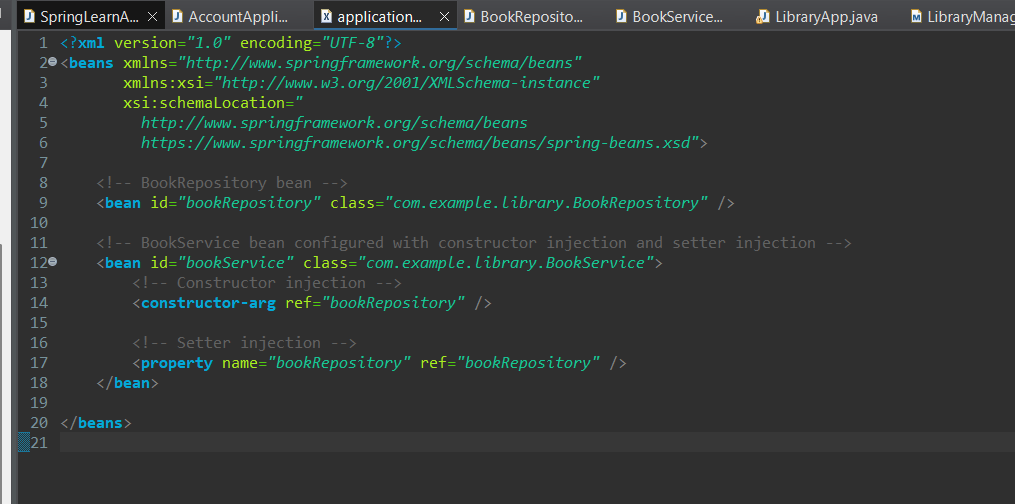
* The Spring IoC container correctly instantiated the BookService bean.
* Dependency injection was performed as configured.
* The BookService instance can successfully access its injected BookRepository dependency.

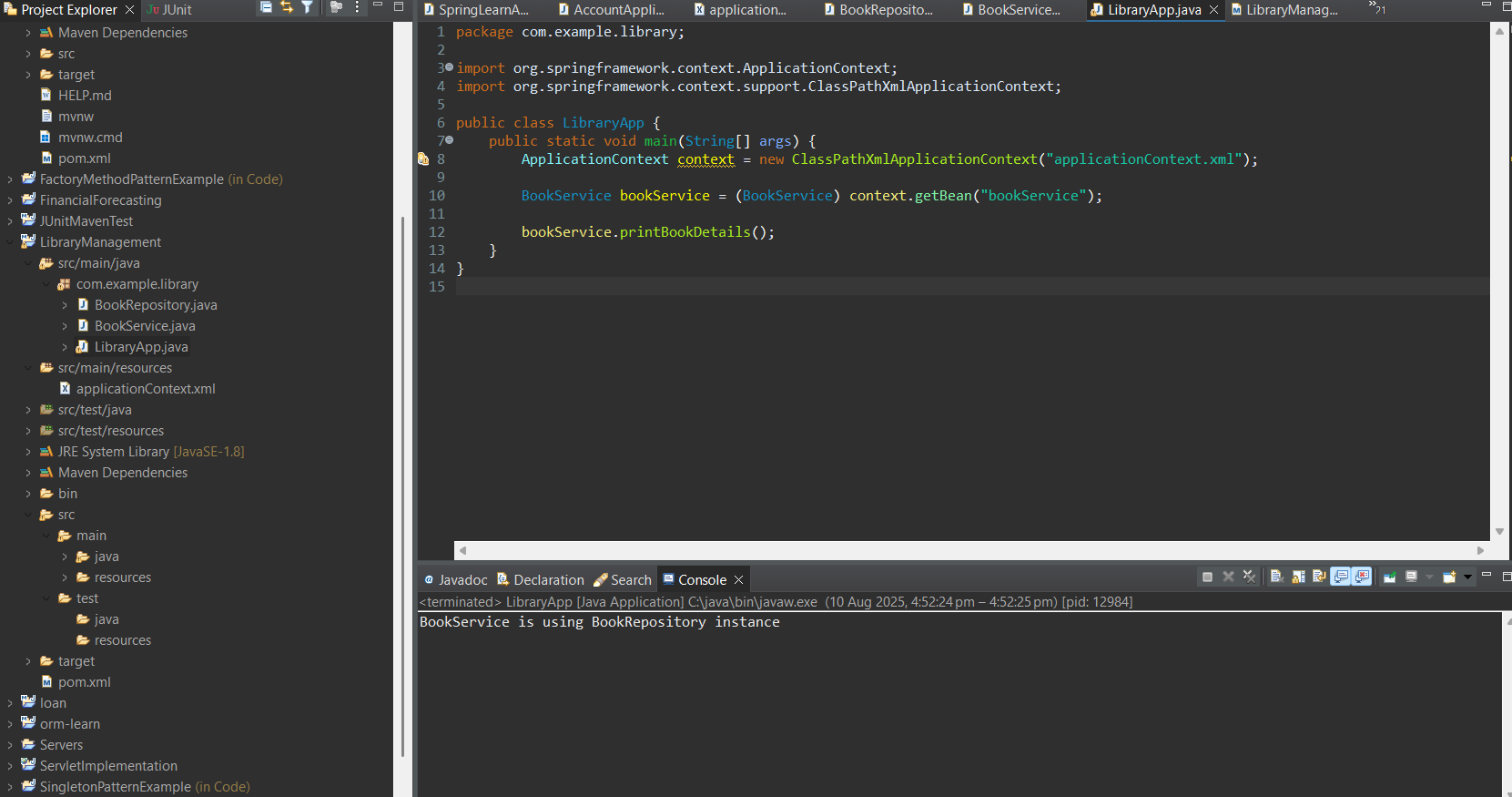
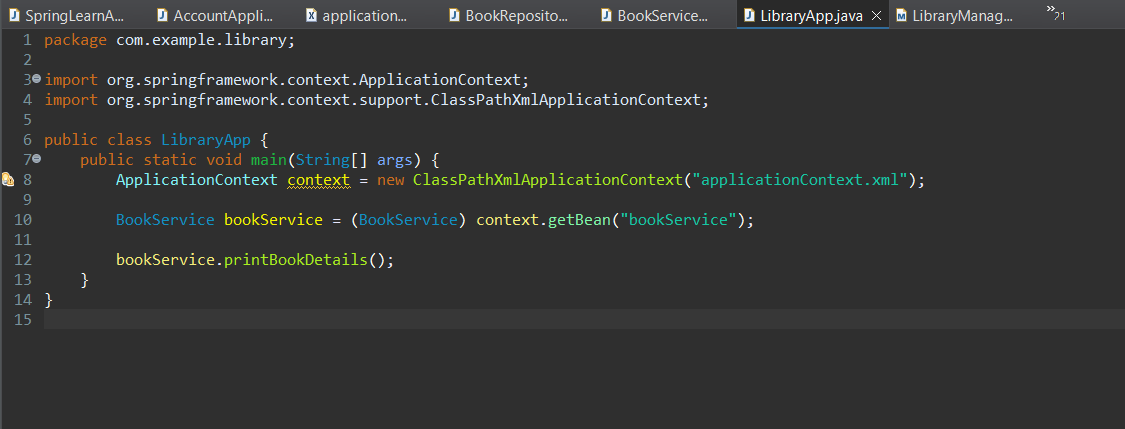
Overall,This exercise successfully demonstrated the application of constructor and setter dependency injection in Spring’s IoC container. It highlights:

* How constructor injection sets dependencies at object creation time.
* How setter injection can override or complement constructor injection post-instantiation.
* The flexibility and power of Spring’s dependency injection mechanisms for managing bean lifecycles and dependencies declaratively.

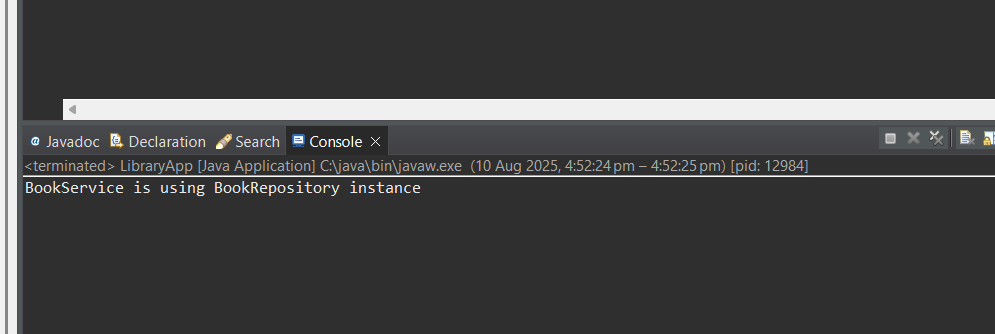
Implementing both methods in a single bean enhances configuration versatility and serves as a practical example for managing dependencies in real-world Spring applications.

****

****

****

**OUTPUT:**



**Exercise 9: Creating a Spring Boot Application**

Scenario:

You need to create a Spring Boot application for the library management system to simplify configuration and deployment.

Steps:

1. Create a Spring Boot Project:

o Use Spring Initializr to create a new Spring Boot project named LibraryManagement.

2. Add Dependencies:

o Include dependencies for Spring Web, Spring Data JPA, and H2 Database.

3. Create Application Properties:

o Configure database connection properties in application.properties.

4. Define Entities and Repositories:

o Create Book entity and BookRepository interface.

5. Create a REST Controller:

o Create a BookController class to handle CRUD operations.

6. Run the Application:

o Run the Spring Boot application and test the REST endpoints.

**IMPLEMENTATION:**

This exercise focused on architecting and implementing a Spring Boot-based microservice to facilitate efficient management of library resources. The objective was to leverage Spring Boot’s rapid application development features to streamline configuration, minimize boilerplate code, and enable seamless integration of persistence and web layers.

The initial phase involved project scaffolding via Spring Initializr, generating a Maven-based Spring Boot project named **LibraryManagement**. The project targeted Java 17 to utilize the latest language features and maintain long term support. Essential dependencies were incorporated, including **Spring Web** for building RESTful APIs, **Spring Data JPA** to abstract and simplify database interactions, and the **H2 in memory database** to provide a lightweight, transient datastore ideal for rapid development cycles and integration testing.

Dependency management was verified through the pom.xml file, ensuring correct versions and scopes to prevent classpath conflicts. The application’s configuration was externalized in application.properties, where datasource parameters—such as the JDBC URL, driver class, username, and password—were explicitly defined. Enabling the H2 console facilitated direct SQL query execution and schema inspection during runtime. Additionally, JPA properties were tuned to specify Hibernate dialect compatibility with H2 and enable verbose SQL statement logging for diagnostic purposes.

At the domain layer, the Book entity was meticulously modeled using JPA annotations, with primary key management delegated via the @Id annotation. Fields including title and author were mapped to corresponding database columns, supporting the persistence of book metadata. The data access layer was encapsulated by the BookRepository interface, extending Spring Data JPA’s JpaRepository. This extension provided out-of-the-box CRUD operations and query derivation capabilities without manual DAO implementation, significantly reducing development effort.

The service layer was exposed through a RESTful controller, BookController, annotated with @RestController and base-mapped to /books. This controller defined endpoints for:

* **GET /books**: Retrieving a complete list of books.
* **GET /books/{id}**: Fetching an individual book by its unique identifier, with proper handling of optional absence through HTTP 404 responses.
* **POST /books**: Persisting new book records via JSON payloads deserialized into Book instances.

Integration of repository beans was achieved via constructor injection, promoting immutability and facilitating unit testing. The REST endpoints adhered to REST best practices, leveraging HTTP status codes and content negotiation.

Upon launching, the Spring Boot application initialized an embedded Tomcat servlet container and auto-configured the Spring context. Hibernate’s EntityManagerFactory was set up seamlessly, managing JPA sessions and transactions. The H2 console was accessible for runtime data validation. Subsequent testing of the REST endpoints, using tools such as Postman and curl, confirmed expected behavior, validating the end-to-end data flow from client requests through the service and persistence layers back to the client.

This exercise demonstrated the power of Spring Boot’s convention-over-configuration paradigm, highlighting how modular dependency injection, auto-configuration, and starter dependencies synergize to accelerate enterprise-grade application development. The design facilitates scalability, maintainability, and future extensibility, making it a robust foundation for complex library management solutions or analogous CRUD-based systems.

Overall this handson’s objective was achieved and can be visualised in below outputs.

**OUTPUT:**

