## **Exercise 1: Configuring a Basic Spring Application**

### **Step 1: Set Up a Spring Project**

**1.1 Create a Maven Project:**

* Use your preferred IDE or command line to create a new Maven project named LibraryManagement.

**1.2 Add Spring Core Dependencies:**

* Open the pom.xml file and add the following dependencies:

XML

1. <dependencies>
2. <dependency>
3. <groupId>org.springframework</groupId>
4. <artifactId>spring-core</artifactId>

<version>6.0.12</version>

1. </dependency>
2. <dependency>
3. <groupId>org.springframework</groupId>
4. <artifactId>spring-context</artifactId>
5. <version>6.0.12</version>
6. </dependency>

</dependencies>

### **Step 2: Configure the Application Context**

**2.1 Create XML Configuration File:**

* Create a file named applicationContext.xml in the src/main/resources directory.

**2.2 Define Beans:**

* Add the following content to applicationContext.xml:

XML

1. <?xml version="1.0" encoding="UTF-8"?>
2. <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.service.BookService">

1. <property name="bookRepository" ref="bookRepository"/>
2. </bean>
3. <bean id="bookRepository" class="com.library.repository.BookRepository"/>
4. </beans>

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

**3.1 Create Service Class:**

* Create a package com.library.service and add a class BookService:

Java

1. package com.library.service;
2. import com.library.repository.BookRepository;
3. public class BookService {
4. private BookRepository bookRepository;
5. public void setBookRepository(BookRepository bookRepository) {

this.bookRepository

1. = bookRepository;
2. }
3. // Book service methods will go here
4. }

**3.2 Create Repository Class:**

* Create a package com.library.repository and add a class BookRepository:

Java

1. package com.library.repository;
2. public class BookRepository {
3. // Book repository methods will go here
4. }

### **Step 4: Run the Application**

**4.1 Create Main Class:**

* Create a class MainApp:

Java

1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public
4. class MainApp {
5. public static void main(String[] args) {

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

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

1. // Test the configuration by calling a method on bookService
2. // ...
3. }
4. }

**4.2 Run the Application:**

* Run the MainApp class.

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## **Exercise 2: Implementing Dependency Injection**

### **Step 1: Modify the XML Configuration**

Update the applicationContext.xml file to wire the BookRepository bean into the BookService bean:

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.service.BookService">

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

</bean>

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

</beans>

### **Step 2: Update the BookService Class**

Ensure that the BookService class has a setter method to accept the BookRepository instance:

Java

package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

private BookRepository bookRepository;

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

// Book service methods will go here

}

### **Step 3: Test the Configuration**

Run the MainApp class to verify the dependency injection:

Java

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.library.service.BookService;

public class MainApp {

public static void main(String[] args) {

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

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

// Test the configuration by calling a method on bookService

// ...}}

While running the MainApp class, the Spring container will create instances of BookService and BookRepository. It will then inject the BookRepository instance into the BookService using the setter injection mechanism.

## **Exercise 3: Implementing Logging with Spring AOP**

### **Step 1: Add Spring AOP Dependency**

Update your pom.xml file to include the Spring AOP dependency:

XML

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>6.0.12</version>

</dependency>

### **Step 2: Create an Aspect for Logging**

Create a package com.library.aspect and add a class LoggingAspect:

Java

package com.library.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

@Aspect

public class LoggingAspect {

private static final Logger logger = LoggerFactory.getLogger(LoggingAspect.class);

@Around("execution(\*com.library.service.\*.\*(..))")

public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {

long start = System.currentTimeMillis();

Object proceed = joinPoint.proceed();

long executionTime = System.currentTimeMillis() - start;

logger.info("Method {} took {} ms to execute", joinPoint.getSignature().getName(), executionTime);

return proceed;

}

}

### **Step 3: Enable AspectJ Support**

Update your applicationContext.xml to enable AspectJ support and register the aspect

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

### **Step 1: Create a New Maven Project**

* **Using an IDE:** Most IDEs (Eclipse, IntelliJ, etc.) have built-in support for creating Maven projects. Use the project wizard to create a new Maven project named LibraryManagement.
* **Command Line:** Use the mvn archetype:generate command to create the project.

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

Open the pom.xml file and add the following dependencies:

XML

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>6.0.12</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>6.0.12</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>6.0.12</version>

</dependency>

</dependencies>

### **Step 3: Configure Maven Plugins**

Add the Maven Compiler Plugin configuration to the pom.xml file

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

### **Step 1: Create Spring Configuration File**

Create an XML configuration file named applicationContext.xml in the src/main/resources directory. Add the following content:

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.service.BookService">

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

</bean>

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

</beans>

### **Step 2: Update the BookService Class**

Ensure that the BookService class has a setter method for BookRepository:

Java

package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

private BookRepository bookRepository;

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository

= bookRepository;

}

// Book service methods will go here

}

### **Step 3: Run the Application**

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

Java

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.library.service.BookService;

public class MainApp {

public static void main(String[] args) {

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

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

// Test the configuration by calling a method on bookService

// ...

}

}

**Explanation:**

* The applicationContext.xml file defines the beans (objects) that Spring will manage and their dependencies.
* The BookService class has a setter method to receive the BookRepository instance.
* The MainApp class loads the Spring context, retrieves the BookService bean, and can then use it.

By running the MainApp class, you will see how Spring injects the BookRepository into the BookService automatically, demonstrating the power of the Spring IoC container.

## **Exercise 6: Configuring Beans with Annotations**

### **Step 1: Enable Component Scanning**

Update your applicationContext.xml file to include component scanning for the com.library package:

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"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

<context:component-scan

base-package="com.library"/>

</beans>

### **Step 2: Annotate Classes**

Annotate the BookService class with @Service and the BookRepository class with @Repository:

Java

package com.library.service;

import org.springframework.stereotype.Service;

import com.library.repository.BookRepository;

@Service

public class BookService {

private BookRepository bookRepository;

// Setter method for bookRepository

}

Java

package com.library.repository;

import org.springframework.stereotype.Repository;

@Repository

public class BookRepository {

// Repository methods

}

### **Step 3: Test the Configuration**

Run the MainApp class to verify the annotation-based configuration:

Java

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.library.service.BookService;

public class MainApp {

public static void main(String[] args) {

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

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

// Test the configuration by calling a method on bookService

// ...

}

}

**Explanation:**

* The @Component annotation is a generic stereotype for any Spring-managed component.
* @Service, @Repository, and @Controller are specializations of @Component for specific layers.
* The context:component-scan element tells Spring to search for classes annotated with @Component and its stereotypes in the specified package.

By using annotations, you've simplified the configuration compared to XML-based configuration. Spring will automatically detect and create beans for the annotated classes.

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

### **Step 1: Configure Constructor Injection**

Update your applicationContext.xml to configure constructor injection for BookService:

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"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"> <context:component-scan

base-package="com.library"/>

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

<constructor-arg ref="bookRepository"/>

</bean>

</beans>

### **Step 2: Configure Setter Injection**

Ensure the BookService class has a setter method for BookRepository:

Java

package com.library.service;

import com.library.repository.BookRepository;

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

import org.springframework.stereotype.Service;

@Service

public class BookService {

private BookRepository bookRepository;

public BookService(BookRepository bookRepository)

{

this.bookRepository = bookRepository;

}

@Autowired

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

// Book service methods

}

### **Step 3: Test the Injection**

Run the MainApp class to verify both constructor and setter injection:

Java

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.library.service.BookService;

public class MainApp {

public static void main(String[] args) {

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

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

// Test the configuration by calling a method on bookService

// ...

}

}

**Explanation:**

* **Constructor Injection:** The constructor-arg element in the applicationContext.xml file injects the bookRepository bean into the BookService constructor.
* **Setter Injection:** The @Autowired annotation on the setter method automatically injects the bookRepository bean into the BookService after the bean is created.

Both constructor and setter injection are used in this example. The constructor injection is mandatory, while the setter injection is optional.

## **Exercise 8: Implementing Basic AOP with Spring**

### **Step 1: Define an Aspect**

Create a package com.library.aspect and add a class LoggingAspect:

Java

package com.library.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

@Aspect

public class LoggingAspect {

private static final Logger logger = LoggerFactory.getLogger(LoggingAspect.class);

@Before("execution(\* com.library.service.\*.\*(..))")

public void logMethodEntry(JoinPoint joinPoint) {

logger.info("Entering method: {}", joinPoint.getSignature());

}

@After("execution(\* com.library.service.\*.\*(..))")

public void logMethodExit(JoinPoint joinPoint) {

logger.info("Exiting method: {}", joinPoint.getSignature());

}

}

### **Step 2: Create Advice Methods**

The LoggingAspect class defines two advice methods:

* logMethodEntry: Executed before a method in the com.library.service package is called.
* logMethodExit: Executed after a method in the com.library.service package returns.

### **Step 3: Configure the Aspect**

Update your applicationContext.xml to register the aspect and enable AspectJ auto-proxying:

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"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<context:component-scan base-package="com.library"/>

<aop:aspectj-autoproxy/>

</beans>

### **Step 4: Test the Aspect**

Run your MainApp class and call methods in your BookService. You should see log messages indicating method entry and exit.

**Explanation:**

* The @Aspect annotation marks the LoggingAspect class as an aspect.
* The @Before and @After annotations define advice methods that will be executed before and after method execution, respectively.
* The pointcut expression execution(\* com.library.service.\*.\*(..)) specifies that the advice should apply to all methods in the com.library.service package.
* The aop:aspectj-autoproxy element enables Spring AOP to create proxies for beans and weave aspects into them.

This exercise demonstrates basic AOP concepts using Spring. You can extend this by adding more advice types (afterReturning, afterThrowing, around) and defining more complex pointcuts.

## **Exercise 9: Creating a Spring Boot Library Management Application**

This exercise guides you through creating a Spring Boot application for your library management system. Spring Boot simplifies configuration and deployment compared to traditional Spring applications.

### **Step 1: Create a Spring Boot Project**

1. **Use Spring Initializr:** Navigate to<https://start.spring.io/>.
2. **Project Setup:**
   * Choose a project name like LibraryManagement.
   * Select Java as the language.
   * Click "Dependencies" and add the following dependencies:
     + Spring Web
     + Spring Data JPA
     + H2 Database (an in-memory database for development)
3. **Generate Project:** Click "Generate" to download the project as a ZIP file.

### **Step 2: Add Dependencies**

Spring Boot includes many dependencies by default. However, you might need additional ones for specific functionalities. Here's an example pom.xml with the chosen dependencies:

XML

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

</dependencies>

### **Step 3: Configure Application Properties**

Create a file named application.properties in the src/main/resources directory. This file holds configuration properties for your application:

Properties

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.jpa.hibernate.ddl-auto=update

# Adjust for automatic schema creation

* spring.datasource.url: Sets the connection URL for the H2 database.
* spring.datasource.driverClassName: Specifies the JDBC driver for H2.
* spring.jpa.hibernate.ddl-auto: Configures how Hibernate creates/updates the database schema (update in this case).

**Note:** Consider using a more robust database solution (e.g., MySQL, PostgreSQL) in production.

### **Step 4: Define Entities and Repositories**

**1. Book Entity:**

Create a class named Book in a package (e.g., com.library.model) to represent a book:

Java

package com.library.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private String author;

private int publicationYear;

// Getters and setters omitted for brevity

}

**2. BookRepository Interface:**

Create an interface named BookRepository in the same package (com.library.model) to interact with the Book entity:

Java

package com.library.model;

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

public interface BookRepository extends JpaRepository<Book, Long> {

}

The JpaRepository interface provides basic CRUD (Create, Read, Update, Delete) operations for the Book entity.

### **Step 5: Create a REST Controller**

Create a class named BookController in a package (e.g., com.library.controller) to handle REST API requests:

Java

package com.library.controller;

import com.library.model.Book;

import com.library.model.BookRepository;

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

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

public

class BookController {

@Autowired

private BookRepository bookRepository;

@GetMapping("/books")

public List<Book> getAllBooks() {

return bookRepository.findAll();

}

// Add more methods for CRUD operations (POST, PUT, DELETE)

}

This example demonstrates a GET request handler (getAllBooks) to retrieve all books. Spring Boot automatically maps controller methods to corresponding HTTP endpoints based on annotations like @GetMapping.