Exercise 1: Configuring a Basic Spring Application

Scenario

You are tasked with developing a web application for managing a library using the Spring Framework.

Steps

1. **Set Up a Spring Project**:
    - Create a Maven project named LibraryManagement.
    - Add Spring Core dependencies in the pom.xml file:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.10</version>
    </dependency>
</dependencies>
```

2. **Configure the Application Context**:
    - Create an XML configuration file named applicationContext.xml in the src/main/resources directory:

```
<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">

    <!-- Define the repository bean -->
    <bean id="bookRepository"
class="com.library.repository.BookRepository"/>

    <!-- Define the service bean and inject the repository -->
    <bean id="bookService" class="com.library.service.BookService">
        <property name="bookRepository" ref="bookRepository"/>
    </bean>
</beans>
```

3. **Define Service and Repository Classes**:
    - Create a package `com.library.service` and add a class `BookService`:

```
package com.library.service;

import com.library.repository.BookRepository;

public class BookService {
    private BookRepository bookRepository;

    public void setBookRepository(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }
}
```

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

```
package com.library.repository;

public class BookRepository {
    // Repository logic
}
```

4. **Run the Application**:
    - Create a main class to load the Spring context and test the configuration:

```
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 =
context.getBean(BookService.class);
        System.out.println("Spring Context Loaded and BookService
Bean Initialized");
    }
}
```

Exercise 2: Implementing Dependency Injection

Scenario

You need to manage the dependencies between the `BookService` and `BookRepository` classes using Spring's IoC and DI.

Steps

1. **Modify the XML Configuration:**
    - Update `applicationContext.xml` to wire `BookRepository` into `BookService`:

```
<bean id="bookService" class="com.library.service.BookService">
    <property name="bookRepository" ref="bookRepository"/>
</bean>
```

2. **Update the BookService Class**:
    - Ensure that the `BookService` class has a setter method for `BookRepository` (as shown in Exercise 1).

3. **Test the Configuration**:
    - Run the `LibraryManagementApplication` main class to verify the dependency injection.

Exercise 3: Implementing Logging with Spring AOP

Scenario

The library management application requires logging capabilities to track method execution times.

Steps

1. **Add Spring AOP Dependency**:
    - Update `pom.xml` to include Spring AOP dependency:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.3.10</version>
```

```
</dependency>
```

2. **Create an Aspect for Logging**:
    - Create a package `com.library.aspect` and add a class `LoggingAspect`:

```
package com.library.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class LoggingAspect {
    @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;
        System.out.println(joinPoint.getSignature() + " executed in
" + executionTime + "ms");
        return proceed;
    }
}
```

3. **Enable AspectJ Support**:
    - Update `applicationContext.xml` to enable AspectJ support and register the aspect:

```
<aop:aspectj-autoproxy/>

<bean id="loggingAspect" class="com.library.aspect.LoggingAspect"/>
```

4. **Test the Aspect**:
    - Run the `LibraryManagementApplication` main class and observe the console for log messages indicating method execution times.

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**:
   - Create a new Maven project named `LibraryManagement`.

5. **Add Spring Dependencies in `pom.xml`**:
   - Include dependencies for Spring Context, Spring AOP, and Spring WebMVC:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.10</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
        <version>5.3.10</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.10</version>
    </dependency>
</dependencies>
```

6. **Configure Maven Plugins**:
   - Configure the Maven Compiler Plugin for Java version 1.8 in the `pom.xml` file:

```
<build>
    <plugins>
        <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>
```

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**:
   - Create an XML configuration file named `applicationContext.xml` in the `src/main/resources` directory (as shown in Exercise 1).

7. **Define Beans**:
   - Define beans for `BookService` and `BookRepository` in the XML file.
8. **Update the BookService Class**:
   - Ensure that the `BookService` class has a setter method for `BookRepository`.
9. **Run the Application**:
   - Create a main class to load the Spring context and test the configuration (as shown in Exercise 1).

Exercise 6: Configuring Beans with Annotations

Scenario

You need to simplify the configuration of beans in the library management application using annotations.

Steps

1. **Enable Component Scanning**:
   - Update `applicationContext.xml` to include component scanning for the `com.library` package:

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

10. **Annotate Classes**:
    - Use `@Service` annotation for the `BookService` class:

```java
import org.springframework.stereotype.Service;

@Service
public class BookService {
    // Service logic
}
```

    - Use `@Repository` annotation for the `BookRepository` class:

```java
import org.springframework.stereotype.Repository;

@Repository
public class BookRepository {
    // Repository logic
}
```

11. **Test the Configuration**:
    - Run the `LibraryManagementApplication` main class to verify the annotation-based configuration.

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**:
    - Update `applicationContext.xml` to configure constructor injection for `BookService`:

```xml
<bean id="bookService" class="com.library.service.BookService">
    <constructor-arg ref="bookRepository"/>
</bean>
```

12. **Configure Setter Injection**:
    - Ensure that the BookService class has a setter method for BookRepository (as shown in Exercise 1).
    - Update the applicationContext.xml to configure setter injection:

```
<bean id="bookService" class="com.library.service.BookService">
    <property name="bookRepository" ref="bookRepository"/>
</bean>
```

13. **Test the Injection**:
    - Run the LibraryManagementApplication main class to verify both constructor and setter injection.

Exercise 8: Implementing Basic AOP with Spring

Scenario

The library management application requires basic AOP functionality to separate cross-cutting concerns like logging and transaction management.

Steps

1. **Define an Aspect**:
    - Create a package com.library.aspect and add a class LoggingAspect (as shown in Exercise 3).

14. **Create Advice Methods**:
    - Define advice methods in LoggingAspect for logging before and after method execution:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.After;

@Aspect
public class LoggingAspect {
    @Before("execution(* com.library.service.*.*(..))")
    public void logBefore() {
        System.out.println("Method execution start");
    }

    @After("execution(* com.library.service.*.*(..))")
    public void logAfter() {
        System.out.println("Method execution end");
    }
}
```

15. **Configure the Aspect**:
    - Update `applicationContext.xml` to register the aspect and enable AspectJ auto-proxying (as shown in Exercise 3).
5. **Test the Aspect**:
    - Run the `LibraryManagementApplication` main class to verify the AOP functionality.

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**:
    - Use Spring Initializr to create a new Spring Boot project named `LibraryManagement`.
2. **Add Dependencies**:
    - Include dependencies for Spring Web, Spring Data JPA, and H2 Database in `pom.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>
        <scope>runtime</scope>
    </dependency>
```

```xml
</dependencies>
```

3. **Create Application Properties**:
    - Configure database connection properties in `application.properties`:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

4. **Define Entities and Repositories**:
    - Create a Book entity:

```java
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Book {
    @Id
    private Long id;
    private String title;
    private String author;

    // Getters and Setters
}
```

  - Create a `BookRepository` interface:

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

public interface BookRepository extends JpaRepository<Book, Long> {
}
```

5. **Create a REST Controller**:
    - Create a `BookController` class to handle CRUD operations:

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/books")
```

```java
public class BookController {

    @Autowired
    private BookRepository bookRepository;

    @GetMapping
    public List<Book> getAllBooks() {
        return bookRepository.findAll();
    }

    @PostMapping
    public Book createBook(@RequestBody Book book) {
        return bookRepository.save(book);
    }

    // Other CRUD operations
}
```

6. **Run the Application**:
    - Run the Spring Boot application and test the REST endpoints.