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.

1. Set Up a Spring Project:

> Create a Maven project named LibraryManagement.

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

Step 1: Create Maven Project: LibraryManagement

```
<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>5.3.34</version> <!-- Choose latest compatible version -->
```

```
    </dependency>
  </dependencies>
</project>
```

2. Configure the Application Context:

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

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

Step 2: Create 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 with dependency injection -->
  <bean id="bookService" class="com.library.service.BookService">
    <property name="bookRepository" ref="bookRepository"/>
  </bean>

</beans>
```

3. Define Service and Repository Classes:

Step 3: Java Classes

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

```
package com.library.repository;


public class BookRepository {
   public void saveBook(String bookName) {
      System.out.println("BookRepository: Saving book - " + bookName);
   }
}
```

> 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;


   // Setter for Dependency Injection
   public void setBookRepository(BookRepository bookRepository) {
      this.bookRepository = bookRepository;
   }
```

```
    public void addBook(String bookName) {
        System.out.println("BookService: Adding book - " + bookName);
        bookRepository.saveBook(bookName);
    }
}
```

4. Run the Application

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

Step 4: Main Class to Run Application

```
package com.library.main;

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) {
        // Load Spring context from XML
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

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

        // Test method
        bookService.addBook("Spring Framework Essentials");
```

```
    }
}
```

EXPECTED OUTPUT:

```
BookService: Adding book - Spring Framework Essentials
BookRepository: Saving book - Spring Framework Essentials
```

Exercise 5: Configuring the Spring IoC Container

Scenario:

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

1. Create Spring Configuration File:

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

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

Step 1: 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 with setter injection -->
    <bean id="bookService" class="com.library.service.BookService">
        <property name="bookRepository" ref="bookRepository"/>
    </bean>

</beans>
```

2. Update the BookService Class:

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

Step 2: BookRepository and BookService Classes

package com.library.repository;

```
public class BookRepository {
    public void save(String bookName) {
        System.out.println("BookRepository: Saving book - " + bookName);
    }
}
```

package com.library.service;

```
import com.library.repository.BookRepository;
```

```
public class BookService {
    private BookRepository bookRepository;

    // Setter for Spring to inject the dependency
    public void setBookRepository(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    public void addBook(String bookName) {
        System.out.println("BookService: Adding book - " + bookName);
        bookRepository.save(bookName);
    }
}
```

3. Run the Application:

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

Step 3: Main Class to Load Spring Context

```
package com.library.main;

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");


        // Get the BookService bean from Spring container
        BookService service = context.getBean("bookService", BookService.class);


        // Use the service
        service.addBook("Clean Code by Robert C. Martin");
    }
}
```

EXPECTED OUTPUT:

```
BookService: Adding book - Clean Code by Robert C. Martin
BookRepository: Saving book - Clean Code by Robert C. Martin
```

<u>Exercise 2: Implementing Dependency Injection</u>

<u>Scenario:</u>

In the library management application, you need to manage the dependencies between the BookService and BookRepository classes using Spring's IoC and DI.

1. Modify the XML Configuration:

> Update applicationContext.xml to wire BookRepository into BookService.

Step 1: Modify 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">

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

  <!-- Define BookService Bean with Dependency Injection -->
  <bean id="bookService" class="com.library.service.BookService">
    <property name="bookRepository" ref="bookRepository"/>
  </bean>

</beans>
```

2. Update the BookService Class:

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

Step 2: Update 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 addBook(String title) {
        System.out.println("BookService: Adding book - " + title);
        bookRepository.save(title);
    }
}
```

package com.library.repository;

```
public class BookRepository {
    public void save(String title) {
        System.out.println("BookRepository: Saving book - " + title);
    }
}
```

3. Test the Configuration:

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

Step 3: Run the Application

```
package com.library.main;
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) {
        // Load Spring application context from XML
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");


        // Retrieve BookService bean
        BookService service = context.getBean("bookService", BookService.class);


        // Test the DI configuration
        service.addBook("The Pragmatic Programmer");
    }
}
```

EXPECTED OUTPUT:

```
BookService: Adding book - The Pragmatic Programmer
BookRepository: Saving book - The Pragmatic Programmer
```

Exercise 7: Implementing Constructor and Setter Injection

Scenario:

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

1. Configure Constructor Injection:

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

Step 1: applicationContext.xml – Constructor + Setter Injection

```
<?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 with Constructor and Setter Injection -->
  <bean id="bookService" class="com.library.service.BookService">
    <!-- Constructor Injection -->
    <constructor-arg value="Library Service v1"/>

    <!-- Setter Injection -->
    <property name="bookRepository" ref="bookRepository"/>
  </bean>

</beans>
```

2. Configure Setter Injection:

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

Step 2: Update BookService Class

```
package com.library.service;
import com.library.repository.BookRepository;
public class BookService {

    private String serviceName;  // Constructor injected
    private BookRepository bookRepository;  // Setter injected

    // Constructor for serviceName
    public BookService(String serviceName) {
        this.serviceName = serviceName;
    }

    // Setter for BookRepository
    public void setBookRepository(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    public void addBook(String title) {
        System.out.println(serviceName + " - Adding book: " + title);
        bookRepository.save(title);
    }
}
```

```
package com.library.repository;


public class BookRepository {
    public void save(String title) {
        System.out.println("BookRepository: Saving book - " + title);
    }
}
```

3. Test the Injection:

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

Step 3: Test the Injection with LibraryManagementApplication.java

```
package com.library.main;


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) {
        // Load Spring Context
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");


        // Retrieve the bean
        BookService bookService = context.getBean("bookService", BookService.class);
```

```
        // Test method
        bookService.addBook("Refactoring by Martin Fowler");
    }
}
```

EXPECTED OUTPUT:

```
Library Service v1 - Adding book: Refactoring by Martin Fowler
BookRepository: Saving book - Refactoring by Martin Fowler
```

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.

mvn archetype:generate -DgroupId=com.library -DartifactId=LibraryManagement -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

2. Add Spring Dependencies in pom.xml:

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

```
<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 Context -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.34</version>
  </dependency>

  <!-- Spring AOP -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.3.34</version>
  </dependency>

  <!-- Spring WebMVC -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
```

```
        <version>5.3.34</version>
    </dependency>

    <!-- Servlet API (provided) -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

3. Configure Maven Plugins:

> Configure the Maven Compiler Plugin for Java version 1.8 in the pom.xml file.

```
<!-- Compiler Plugin Configuration -->
<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>
</project>
```

Once pom.xml is ready, you can build the project using:

mvn clean install

- This will download dependencies and compile the project using Java 1.8.

Exercise 9: Creating a Spring Boot Application

SCENARIO:

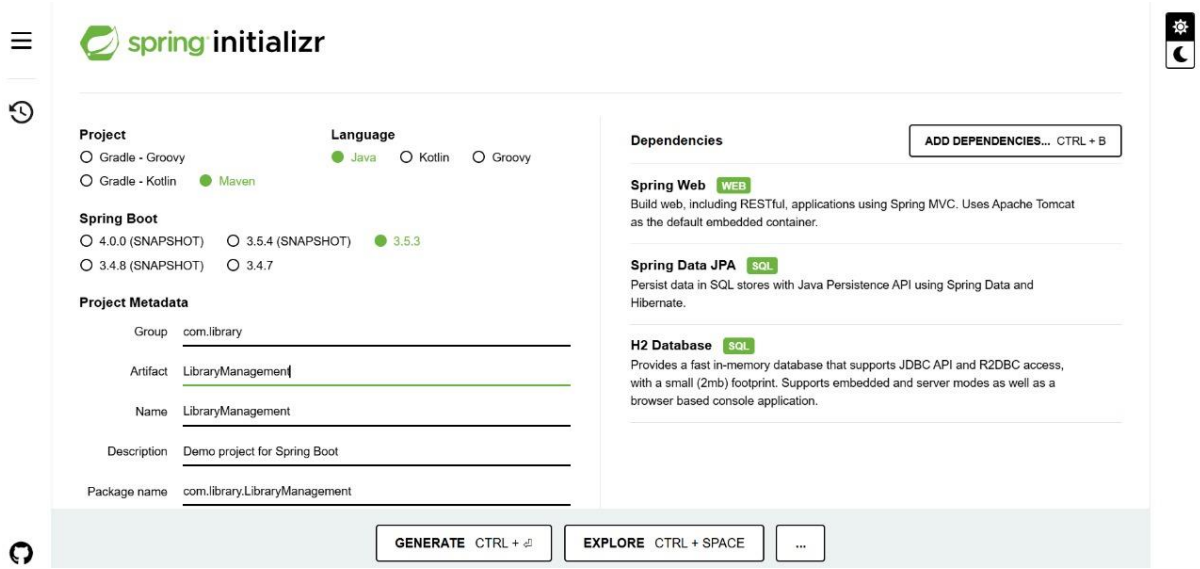
You need to create a Spring Boot application for the library management system to simplify configuration and deployment.

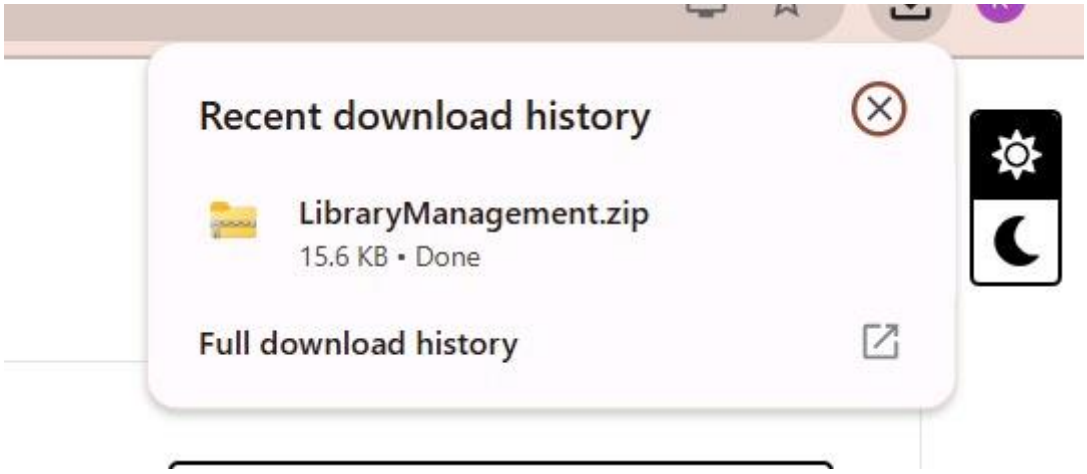
1. Create a Spring Boot Project:

> 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.





3. Create Application Properties:

> Configure database connection properties in application.properties.

```
# H2 in-memory DB config
spring.datasource.url=jdbc:h2:mem:librarydb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
```

```
spring.datasource.password=

# JPA and Hibernate
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

4. Define Entities and Repositories:

> Create Book entity and BookRepository interface.

```
package com.library.entity;

import jakarta.persistence.*;

@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private String author;

    // Getters and Setters
    public Long getId()
```

```
   {
  return id;
 }
 public void setId(Long id)
{
 this.id = id;
}

  public String getTitle()
{
 return title;
 }
   public void setTitle(String title)
{
 this.title = title;
 }

   public String getAuthor()
{
 return author;
 }
   public void setAuthor(String author)
{
 this.author = author;
 }}
```

package com.library.repository;

```
import com.library.entity.Book;
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.

```
package com.library.controller;

import com.library.entity.Book;
import com.library.repository.BookRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/books")
public class BookController {

    @Autowired
    private BookRepository bookRepository;



    @PostMapping
```

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

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

@GetMapping("/{id}")
public Book getBookById(@PathVariable Long id) {
    return bookRepository.findById(id).orElse(null);
}

@PutMapping("/{id}")
public Book updateBook(@PathVariable Long id, @RequestBody Book updatedBook) {
    Book book = bookRepository.findById(id).orElse(null);
    if (book != null) {
        book.setTitle(updatedBook.getTitle());
        book.setAuthor(updatedBook.getAuthor());
        return bookRepository.save(book);
    }
    return null;
}

@DeleteMapping("/{id}")
public void deleteBook(@PathVariable Long id) {
    bookRepository.deleteById(id);
```

```
    }
}
```

6. Run the Application:

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

```
package com.library;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class LibraryManagementApplication {
    public static void main(String[] args) {
        SpringApplication.run(LibraryManagementApplication.class, args);
    }
}
```

- **POST** /books

```
json
{
  "title": "Effective Java",
  "author": "Joshua Bloch"
}
```

- **GET** /books
  Returns list of all books.
- **GET** /books/1
  Returns book with ID 1.

- **PUT** /books/1

json

```
{
  "title": "Clean Code",
  "author": "Robert C. Martin"
}
```

- **DELETE** /books/1

Deletes book with ID 1.

SUPERSET : 6407636
KANMANI MURUGHAIYAN