# Cognizant Digital Nurture 4.0 – Java FSE

## Week 3 – Hands-On Report

Name: Shaik Sulthan - 6431169

Track: Deep Skilling – Java Full Stack Engineer

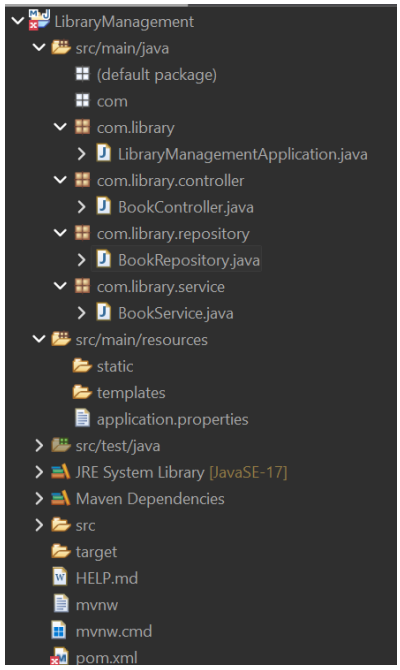
Batch: 2025

# 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.

## Steps:

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

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

```
package com.library.service;

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

```java
import org.springframework.stereotype.Service;
import com.library.repository.BookRepository;

@Service
public class BookService {

    private final BookRepository bookRepository;

    // Constructor-based dependency injection
    @Autowired
    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    public void displayService() {
        System.out.println("BookService: Service is working...");
        bookRepository.displayRepo();
    }
}
```

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

```java
package com.library.repository;

import org.springframework.stereotype.Repository;

@Repository  // ☑ This is required so Spring can register the bean
public class BookRepository {

    public void displayRepo() {
        System.out.println("BookRepository: Fetching book data...");
    }
}
```

**Run the Application:**

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

**http://localhost:8080/books**

Here are your books!

# 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.

## Steps:

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

      **applicationContext.xml :**

```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>
```

2. **Update the BookService Class:**
    - Ensure that **BookService** class has a setter method for **BookRepository**.

      **BookService.java :**

```java
package com.library.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.library.repository.BookRepository;

@Service
```

```
public class BookService {

    private final BookRepository bookRepository;

    @Autowired  // constructor injection
    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    public String displayService() {
        return "BookService: Service is working...\n" + bookRepository.displayRepo();
    }
}
```

**BookRepository.java :**

```
package com.library.repository;

import org.springframework.stereotype.Repository;

@Repository
public class BookRepository {

    public String displayRepo() {
        return "BookRepository: Fetching book data...";
    }
}
```

3.**Test the Configuration:**

- Run the **LibraryManagementApplication** main class to verify the dependency injection.

**LibraryManagementApplication.java :**

```
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);
    }
}
```

**OUTPUT:**

BookService: Service is working... BookRepository: Fetching book data...

**http://localhost:8080/books**

# 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.

      **UPDATED pom.xml :**

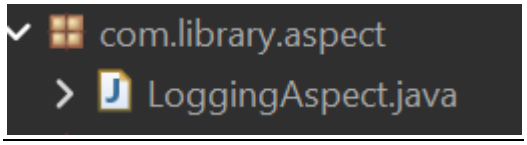
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>3.5.3</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.library</groupId>
        <artifactId>LibraryManagement</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>LibraryManagement</name>
        <description>Demo project for Spring Boot</description>
        <url/>
        <licenses>
                <license/>
        </licenses>
        <developers>
                <developer/>
        </developers>
        <scm>
                <connection/>
                <developerConnection/>
                <tag/>
                <url/>
        </scm>
        <properties>
                <java.version>17</java.version>
        </properties>
        <dependencies>
```

```
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-devtools</artifactId>
			<scope>runtime</scope>
			<optional>true</optional>
		</dependency>
		<dependency>
			<groupId>com.h2database</groupId>
			<artifactId>h2</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>

		</dependency>
		<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>


	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

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

log execution times.

**Package com.library.aspect :**




## LoggingAspect:

```
package com.library.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Around("execution(* com.library.service.*.*(..))")
    public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable
{
        String methodName = joinPoint.getSignature().toShortString();
        long start = System.currentTimeMillis();

        System.out.println(">> [AOP] Starting method: " + methodName);

        Object result = joinPoint.proceed();

        long end = System.currentTimeMillis();
        System.out.println(">> [AOP] Completed method: " + methodName + " in " +
(end - start) + " ms");

        return result;
    }
}
```

### 3. Enable AspectJ Support:

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

**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>
```

4. **Test the Aspect:**
    - Run the

```
>> [AOP] Starting method: BookService.displayService()
>> [AOP] Completed method: BookService.displayService() in 2 ms
>> [AOP] Starting method: BookService.displayService()
>> [AOP] Completed method: BookService.displayService() in 0 ms
```

**LibraryManagementApplication** main class and observe the console for log messages indicating method execution times.

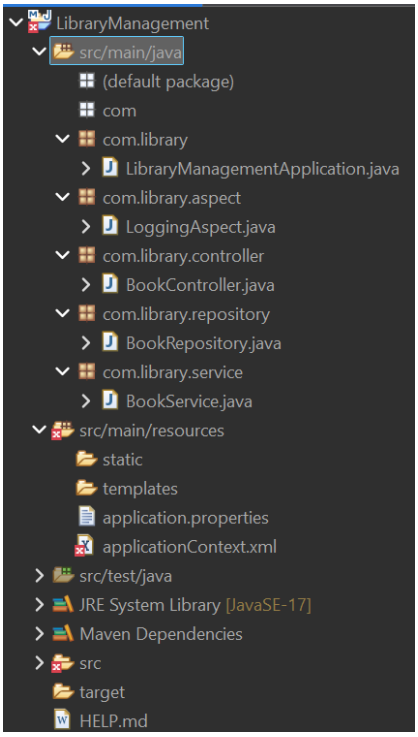
# Exercise 4: Creating and Configuring a Maven Project

## 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**.

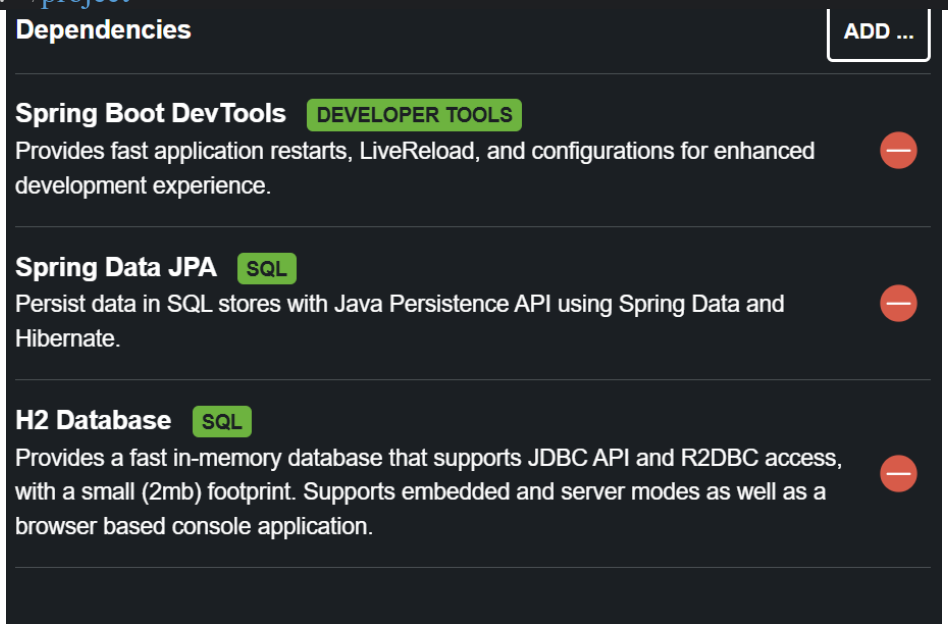



2. **Add Spring Dependencies in pom.xml:**
    - Include dependencies for Spring Context, Spring AOP, and Spring WebMVC.

```
<?xml version="1.0" encoding="UTF-8"?>
<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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
```

```
10.             <version>3.5.3</version>
11.             <relativePath/> <!-- lookup parent from repository -->
12.         </parent>
13.         <groupId>com.library</groupId>
14.         <artifactId>LibraryManagement</artifactId>
15.         <version>0.0.1-SNAPSHOT</version>
16.         <name>LibraryManagement</name>
17.         <description>Demo project for Spring Boot</description>
18.         <url/>
19.         <licenses>
20.             <license/>
21.         </licenses>
22.         <developers>
23.             <developer/>
24.         </developers>
25.         <scm>
26.             <connection/>
27.             <developerConnection/>
28.             <tag/>
29.             <url/>
30.         </scm>
31.         <properties>
32.             <java.version>17</java.version>
33.         </properties>
34.         <dependencies>
35.             <dependency>
36.                 <groupId>org.springframework.boot</groupId>
37.                 <artifactId>spring-boot-starter-data-jpa</artifactId>
38.             </dependency>
39.             <dependency>
40.                 <groupId>org.springframework.boot</groupId>
41.                 <artifactId>spring-boot-starter-web</artifactId>
42.             </dependency>
43.
44.             <dependency>
45.                 <groupId>org.springframework.boot</groupId>
46.                 <artifactId>spring-boot-devtools</artifactId>
47.                 <scope>runtime</scope>
48.                 <optional>true</optional>
49.             </dependency>
50.             <dependency>
51.                 <groupId>com.h2database</groupId>
52.                 <artifactId>h2</artifactId>
53.                 <scope>runtime</scope>
54.             </dependency>
55.             <dependency>
56.                 <groupId>org.springframework.boot</groupId>
57.                 <artifactId>spring-boot-starter-test</artifactId>
58.                 <scope>test</scope>
59.
```

```
60.                 </dependency>
61.                 <dependency>
62.     <groupId>org.springframework.boot</groupId>
63.     <artifactId>spring-boot-starter-aop</artifactId>
64. </dependency>
65.
66.
67.         </dependencies>
68.
69.         <build>
70.                 <plugins>
71.                         <plugin>
72.                                 <groupId>org.springframework.boot</groupId>
73.                                 <artifactId>spring-boot-maven-
    plugin</artifactId>
74.                         </plugin>
75.                 </plugins>
76.         </build>
77.
78. </project>
```




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

      **localhost:8080**

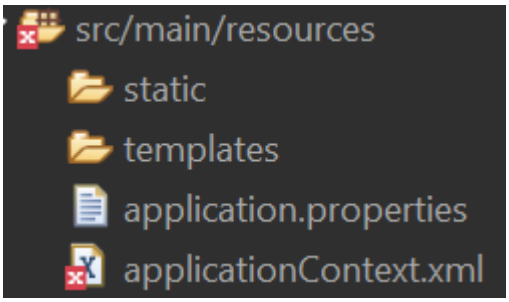
# Exercise 5: Configuring the Spring IoC Container

## 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.




    - Define beans for **BookService** and **BookRepository** in the XML file.

**BEAN's of BookService and BookRepository :**

```
<bean id="bookRepository" class="com.library.repository.BookRepositor

<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**.

**BookService with setter method :**

```
package com.library.service;

import com.library.repository.BookRepository;

public class BookService {
```

```
    private BookRepository bookRepository;

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

    public void displayService() {
        System.out.println("BookService: Service is working...");
        bookRepository.displayRepo();
    }
}
```

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

**Mian calss**

```
package com.library;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.library.service.BookService;

public class LibraryManagementApplication {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

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

BookService: Service is working... BookRepository: Fetching book data...

**http://localhost:8080/books**

# 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.

      **Update applicationContext.xml :**

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

  <!-- Enable annotation scanning in com.library and sub-packages -->
  <context:component-scan base-package="com.library" />

</beans>
```

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

      **@Service annotation in BookService :**

```java
package com.library.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.library.repository.BookRepository;

@Service
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    public void displayService() {
```

```
        System.out.println("BookService: Service is working...");
        bookRepository.displayRepo();
    }
}
```

- Use **@Repository** annotation for the **BookRepository** class.

**@Repository annotation in BookRepository :**

```
package com.library.repository;

import org.springframework.stereotype.Repository;

@Repository
public class BookRepository {

    public void displayRepo() {
        System.out.println("BookRepository: Fetching book data...");
    }
}
```

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

```
BookService: Service is working...
BookRepository: Fetching book data...
```

**OUTPUT from console**

# 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**.

      **Update applicationContext.xml with 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"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

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

    <!-- BookService Bean with constructor + setter injection -->
    <bean id="bookService" class="com.library.service.BookService">
        <!-- Constructor injection -->
        <constructor-arg value="LibraryAppService"/>

        <!-- 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**.

## BookService with a setter method for BookRepository :

```
package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

    private BookRepository bookRepository;
    private String serviceName;

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

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

    public void displayService() {
        System.out.println("[" + serviceName + "] BookService: Service is working...");
        bookRepository.displayRepo();
    }
}
```

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

```
[LibraryAppService] BookService: Service is working...
BookRepository: Fetching book data...
```

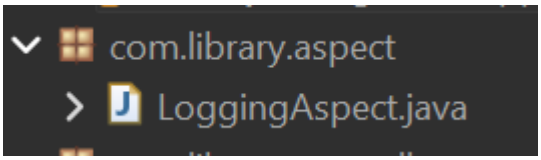
# Exercise 8: Implementing Basic AOP with Spring

## 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**.




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

**LoggingAspect.java :**

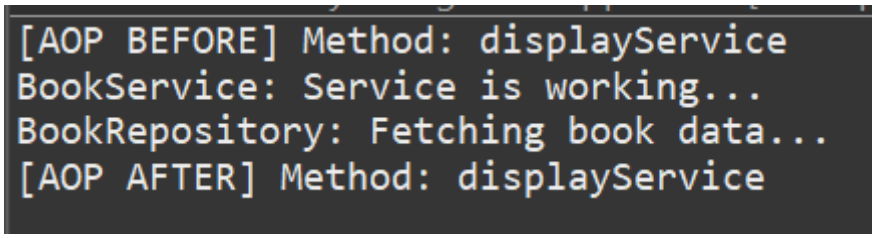
```
package com.library.aspect;

import org.aspectj.lang.JoinPoint;

public class LoggingAspect {

    public void logBefore(JoinPoint joinPoint) {
        System.out.println("[AOP BEFORE] Method: " +
joinPoint.getSignature().getName());
    }

    public void logAfter(JoinPoint joinPoint) {
        System.out.println("[AOP AFTER] Method: " +
joinPoint.getSignature().getName());
    }
}
```

3. **Configure the Aspect:**
    - Update **applicationContext.xml** to register the aspect and enable **AspectJ** auto-proxying.

```
4.          <!-- AspectJ Weaver -->
5.  <dependency>
```

```
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.20.1</version>
</dependency>
```

4. **Test the Aspect:**
    - Run the **LibraryManagementApplication** main class to verify the AOP functionality.



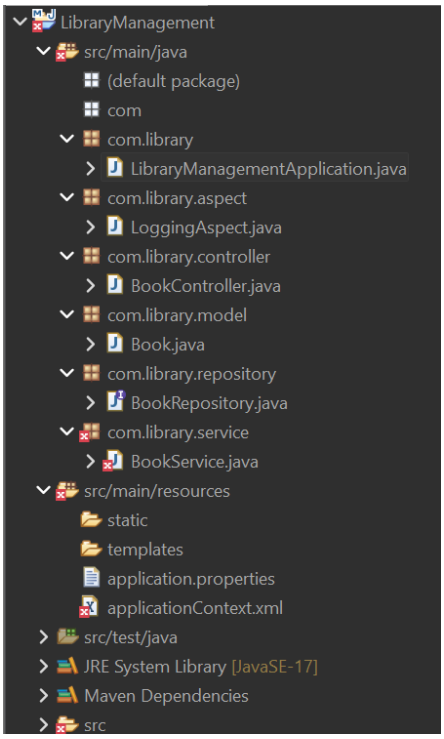
# Exercise 9: Creating a Spring Boot Application

## 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**.

```
<!-- Spring Data JPA -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <!-- Spring Web -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

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


    <!-- H2 Database (in-memory DB for testing) -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
```

23. **Create Application Properties:**

- Configure database connection properties in **application.properties**.

```
# H2 console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# JPA settings
spring.datasource.url=jdbc:h2:mem:librarydb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
```

35. **Define Entities and Repositories:**

- Create **Book** entity and **BookRepository** interface.

**Book :**

```
package com.library.model;

import jakarta.persistence.*;

@Entity
public class Book {

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

    private String title;
    private String author;

    // Getters & 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; }
}
```

**Create a REST Controller:**

- Create a **BookController** class to handle CRUD operations.

```
36. package com.library.controller;
37.
38. import com.library.model.Book;
39. import com.library.repository.BookRepository;
40. import org.springframework.beans.factory.annotation.Autowired;
41. import org.springframework.web.bind.annotation.*;
42.
43. import java.util.List;
44.
45. @RestController
46. @RequestMapping("/books")
47. public class BookController {
48.
49.     @Autowired
50.     private BookRepository bookRepository;
51.
52.     // GET all books
53.     @GetMapping
54.     public List<Book> getAllBooks() {
55.         return bookRepository.findAll();
56.     }
57.
58.     // GET book by ID
59.     @GetMapping("/{id}")
60.     public Book getBookById(@PathVariable Long id) {
61.         return bookRepository.findById(id).orElse(null);
62.     }
63.
64.     // POST create new book
65.     @PostMapping
66.     public Book addBook(@RequestBody Book book) {
67.         return bookRepository.save(book);
68.     }
69.
70.     // PUT update book
71.     @PutMapping("/{id}")
72.     public Book updateBook(@PathVariable Long id, @RequestBody Book
    updatedBook) {
```

```
73.         return bookRepository.findById(id).map(book -> {
74.             book.setTitle(updatedBook.getTitle());
75.             book.setAuthor(updatedBook.getAuthor());
76.             return bookRepository.save(book);
77.         }).orElse(null);
78.     }
79.
80.     // DELETE book
81.     @DeleteMapping("/{id}")
82.     public void deleteBook(@PathVariable Long id) {
83.         bookRepository.deleteById(id);
84.     }
85. }
```

86. **Run the Application:**

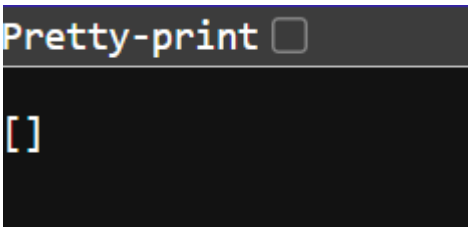
- Run the Spring Boot application and test the REST endpoints.

  GET = List all books

  POST = {"title":"ABC", "author":"XYZ"}

  PUT = Update book

  DELETE = Delete book




http://localhost:8080/books url with zero books