**Week-3 Hands-On Solutions**

**Spring Core And Maven**

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

**Overview:**

This project uses the Spring Core module to manage dependencies between different components of the library system:

* BookRepository: Responsible for data access logic, such as saving a book.
* BookService: Acts as a service layer that processes the business logic and interacts with the repository.
* MainApp: The main driver class that initializes the Spring container and triggers the service methods.

**Spring Configuration:**

* Uses applicationContext.xml to define Spring beans.
* Beans are wired using setter injection.
* The Spring container manages the lifecycle and dependencies of these beans.

**Files structure:**

LibraryManagementSystem/

|-- pom.xml

|-- src/

|-- main/

|-- java/

|-- com/

|-- library/

|-- MainApp.java

|-- repository/

|-- BookRepository.java

|-- service/

|-- BookService.java

|-- resources/

|-- applicationContext.xml

**1.pom.xml:**

<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>LibraryManagementSystem</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <dependencies>  
 <!-- Spring Context (Core + Beans) -->  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.3.33</version>  
 </dependency>  
 </dependencies>  
</project>

**2. BookRepository.java**

package com.library.repository;  
  
public class BookRepository {  
 public void saveBook(String bookName) {  
 System.*out*.println("Book successfully saved to Library Database → [" + bookName.toUpperCase() + "]");  
 }  
}

**3. BookService.java**

package com.library.service;  
  
import com.library.repository.BookRepository;  
  
public class BookService {  
 private BookRepository bookRepository;  
  
 public void setBookRepository(BookRepository bookRepository) {  
 this.bookRepository = bookRepository;  
 }  
  
 public void addBook(String bookName) {  
 System.*out*.println(" Processing request to add book: " + bookName);  
 bookRepository.saveBook(bookName);  
 System.*out*.println("Book addition completed.");  
 }  
}

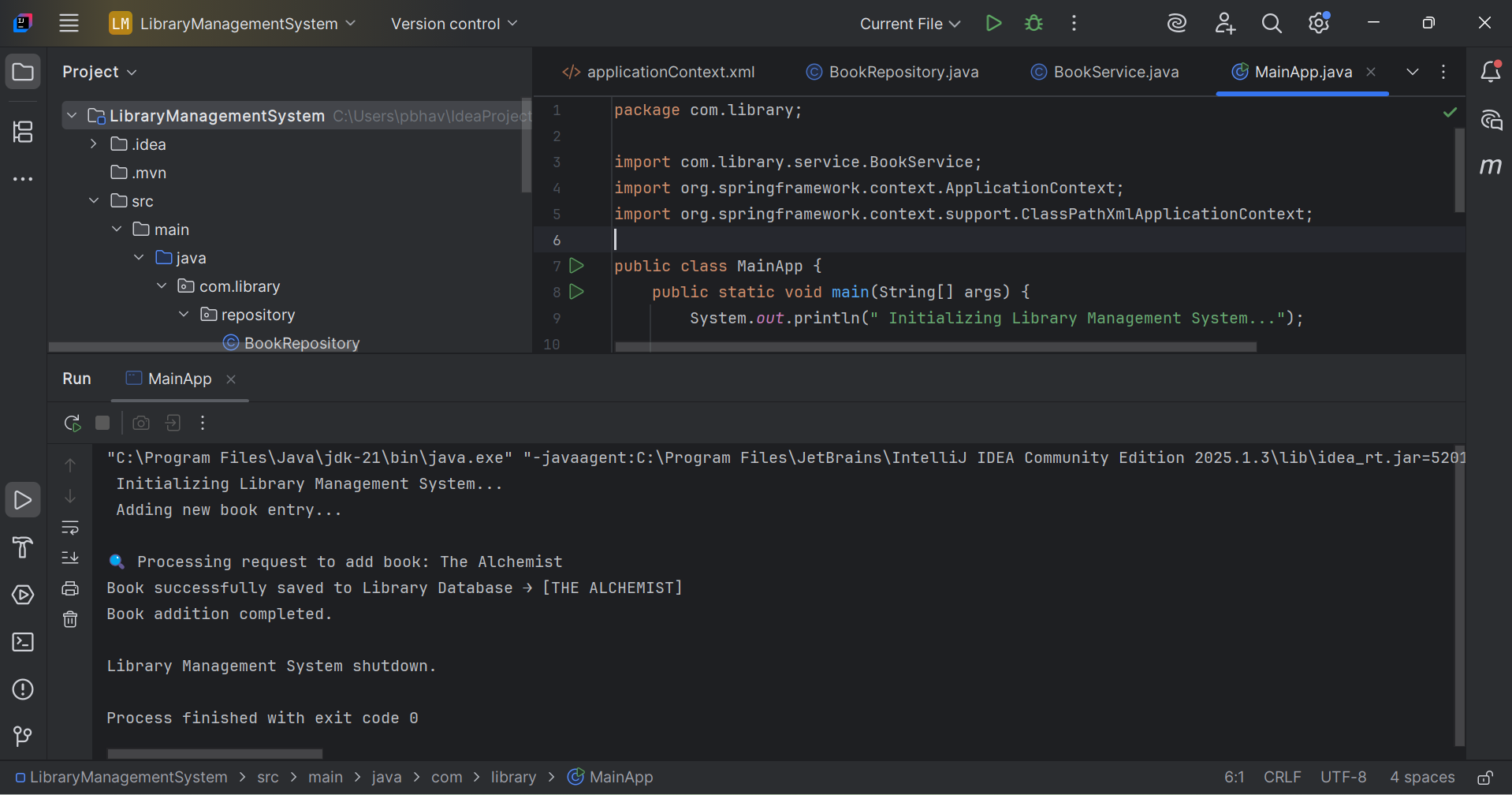
**4.applicationContext.xml**

<?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  
 https://www.springframework.org/schema/beans/spring-beans.xsd">  
  
 <!-- Repository Bean -->  
 <bean id="bookRepository" class="com.library.repository.BookRepository" />  
  
 <!-- Service Bean -->  
 <bean id="bookService" class="com.library.service.BookService">  
 <property name="bookRepository" ref="bookRepository"/>  
 </bean>  
  
</beans>

**5. MainApp.java**

package com.library;  
  
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) {  
 System.*out*.println(" Initializing Library Management System...");  
  
 ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
 BookService bookService = context.getBean("bookService", BookService.class);  
  
 System.*out*.println(" Adding new book entry...\n");  
 bookService.addBook("The Alchemist");  
  
 System.*out*.println("\nLibrary Management System shutdown.");  
 }  
}

**Output:**

****

**Exercise 2: Implementing Dependency Injection**

In this exercise, I worked on implementing Dependency Injection (DI) in my Library Management System project using Spring Framework. The main goal was to manage the dependency between the BookService and BookRepository classes by using Spring's IoC (Inversion of Control) container and DI concepts.

**Objective:**

The objective of this task was to properly configure Dependency Injection using Spring's XML configuration. By doing this, I ensured that BookService and BookRepository are loosely coupled, and Spring takes care of object creation and dependency management, rather than doing it manually in the code.

**Project File Structure:**

LibraryManagementSystem/

|-- pom.xml

|-- src/

|-- main/

|-- java/

|-- com/

|-- library/

|-- LibraryManagementApplication.java

|-- repository/

|-- BookRepository.java

|-- service/

|-- BookService.java

|-- resources/

|-- applicationContext.xml

**Steps I Followed:**

**Updated XML Configuration in**

**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  
 https://www.springframework.org/schema/beans/spring-beans.xsd">  
  
 <!-- Bean for BookRepository -->  
 <bean id="bookRepository" class="com.library.repository.BookRepository" />  
  
 <!-- Bean for BookService with dependency injection of bookRepository -->  
 <bean id="bookService" class="com.library.service.BookService">  
 <property name="bookRepository" ref="bookRepository" />  
 </bean>  
  
</beans>

**2.Added Setter Method in BookService.java**

package com.library.service;  
  
import com.library.repository.BookRepository;  
  
public class BookService {  
  
 private BookRepository bookRepository;  
  
 // Setter method for dependency injection  
 public void setBookRepository(BookRepository bookRepository) {  
 this.bookRepository = bookRepository;  
 }  
  
 public void addBook(String bookName) {  
 System.*out*.println("Starting the process to add book: " + bookName);  
 bookRepository.saveBook(bookName);  
 System.*out*.println("Successfully added the book: " + bookName.toUpperCase());  
 }  
}

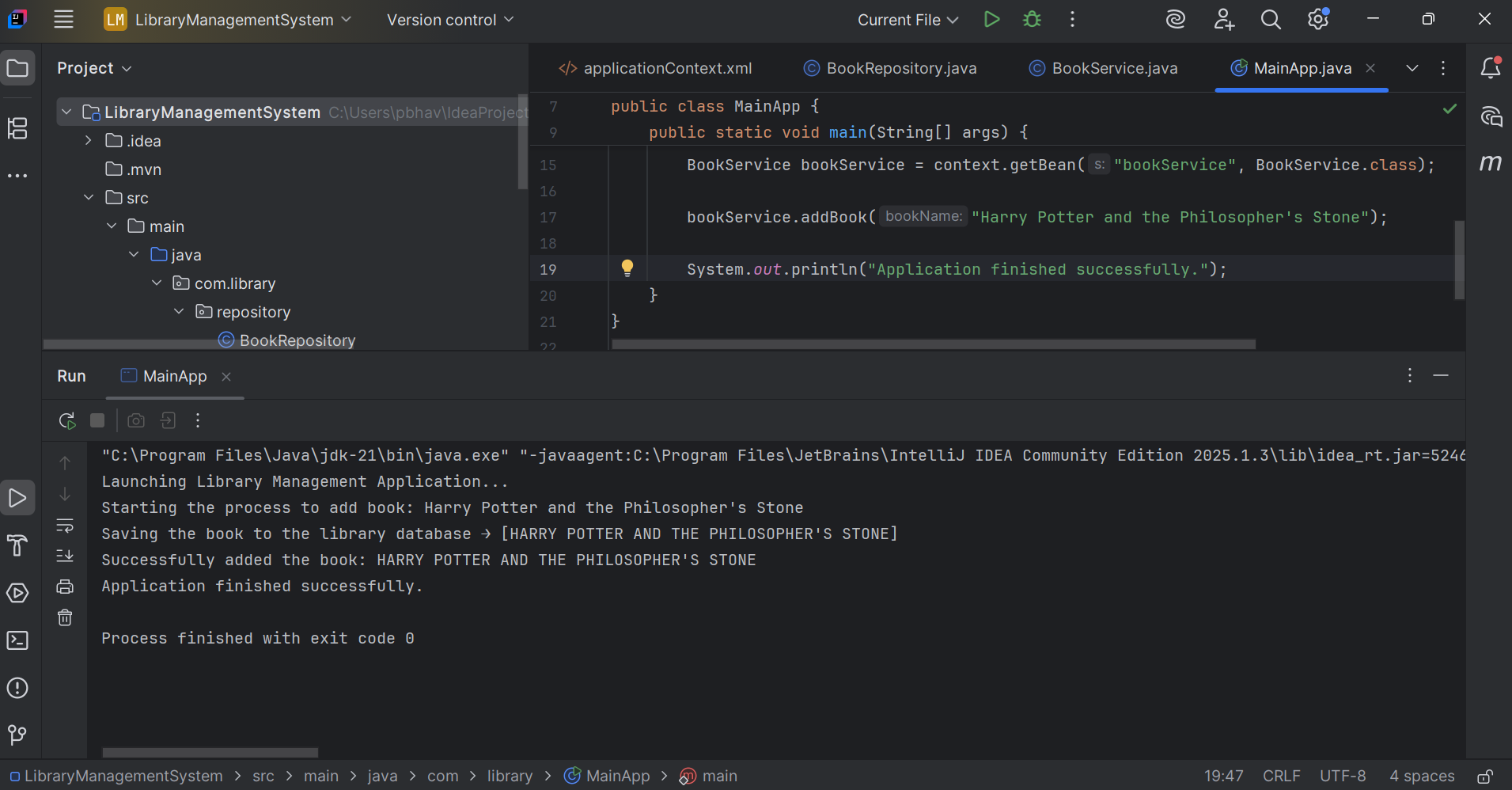
**3.Tested the Application**

To test if Dependency Injection was working properly, I ran my main class:

**MainApp.java**

package com.library;  
  
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) {  
  
 System.*out*.println("Launching Library Management Application...");  
  
 ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
  
 BookService bookService = context.getBean("bookService", BookService.class);  
  
 bookService.addBook("Harry Potter and the Philosopher's Stone");  
  
 System.*out*.println("Application finished successfully.");  
 }  
}

**Output:**



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

In this exercise, I created a new Maven project for my Library Management application. My goal was to set up the project with the necessary Spring dependencies and configure it to use Java 1.8 for compiling the code.

**Objective**

* To create a Maven project named **LibraryManagementSystem**.
* To add essential Spring dependencies: **Spring Context**, **Spring AOP**, and **Spring WebMVC**.
* To configure the Maven Compiler Plugin to ensure the project compiles with Java version 1.8.

**Steps I Followed**

**Step 1: Creating the Maven Project**

I created a new Maven project named LibraryManagementSystem using my IDE. This created the standard Maven folder structure including src/main/java, src/main/resources, and the pom.xml file.

**Structure**

LibraryManagementSystem/

|-- src/

| |-- main/

| | |-- java/

| | |-- resources/

| |-- test/

| |-- java/

|-- pom.xml

**Step 2: Adding Spring Dependencies**

In the **pom.xml** file, I added the following dependencies inside the <dependencies> tag:

**pom.xml:**

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.33</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.33</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.33</version>

</dependency>

These dependencies help me use core Spring features, aspect-oriented programming (AOP), and web MVC capabilities.

**Step 3: Configuring Maven Compiler Plugin**

To make sure my project compiles with Java 1.8, I configured the Maven Compiler Plugin in the <build> section of the pom.xml as follows:

<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>

**Final pom.xml File**

Here is the complete content of my pom.xml file after the changes:

<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>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.33</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.33</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.33</version>

</dependency>

</dependencies>

<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>

**Summary**

By completing this exercise, I successfully created a Maven project and configured it with the necessary Spring dependencies and Java compiler settings. This setup prepares my project for building Spring-based applications and ensures compatibility with Java 1.8.

**Spring Data JPA with Spring Boot, Hibernate**

**Exercise: Spring Data JPA - Quick Example**

**Objective**

The goal of this exercise is to create a Spring Boot application that uses Spring Data JPA to connect with a MySQL database and perform CRUD operations on a custom entity called SpaceMission. This helps me understand how JPA works, how entities map to tables, and how repositories provide database operations.

**Tools and Technologies Used**

* IntelliJ IDEA Ultimate Edition (latest stable version)
* MySQL Server 8.0
* MySQL Workbench 8
* Maven 3.6.2
* Spring Boot 2.7.x (via Spring Initializr)
* Java 1.8

**Steps I Followed in IntelliJ IDEA**

**1. Creating the Project using Spring Initializr**

* Opened IntelliJ IDEA, selected **New Project** > **Spring Initializr**.
* Entered Group: com.spaceagency
* Entered Artifact: mission-control
* Added dependencies:
  + Spring Boot DevTools
  + Spring Data JPA
  + MySQL Driver
* Chose Java 1.8 and Maven as build tool.
* Finished and waited for IntelliJ to generate the project.

**2. Database Preparation**

* Opened MySQL Workbench.

1.Created a schema called missiondb:

CREATE SCHEMA missiondb;

2.Created table space\_mission for my entity:

CREATE TABLE space\_mission (

mission\_id VARCHAR(10) PRIMARY KEY,

mission\_name VARCHAR(100),

launch\_year INT,

status VARCHAR(20)

);

3.Inserted sample data:

INSERT INTO space\_mission VALUES ('MSN001', 'Voyager 1', 1977, 'Active');

INSERT INTO space\_mission VALUES ('MSN002', 'Apollo 11', 1969, 'Completed');

**3. Configuring application.properties**

Configured database connection and logging:

logging.level.org.springframework=info

logging.level.com.spaceagency=debug

logging.level.org.hibernate.SQL=trace

logging.level.org.hibernate.type.descriptor.sql=trace

logging.pattern.console=%d{dd-MM-yy} %d{HH:mm:ss.SSS} %-20.20thread %5p %-25.25logger{25} %25M %4L %m%n

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/missiondb

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.hibernate.ddl-auto=validate

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

**4. Created Entity Class SpaceMission**

* Package: com.spaceagency.missioncontrol.model
* Annotated with @Entity and @Table(name = "space\_mission").
* Mapped fields with @Id and @Column as per database columns.

**5. Created Repository Interface SpaceMissionRepository**

* Package: com.spaceagency.missioncontrol.repository
* Extended JpaRepository<SpaceMission, String>
* Annotated with @Repository

**6. Created Service Class SpaceMissionService**

* Package: com.spaceagency.missioncontrol.service
* Annotated with @Service
* Injected repository with @Autowired
* Defined method getAllMissions() annotated with @Transactional to fetch all missions.

**7. Testing in Main Application Class MissionControlApplication**

* Fetched SpaceMissionService bean from application context.
* Called getAllMissions() and logged results using SLF4J Logger.

**1.SpaceMission.java (Entity Class)**

package com.spaceagency.missioncontrol.model;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "space\_mission")

public class SpaceMission {

@Id

private String missionId;

private String missionName;

private int launchYear;

private String status;

// Getters and Setters

public String getMissionId() { return missionId; }

public void setMissionId(String missionId) { this.missionId = missionId; }

public String getMissionName() { return missionName; }

public void setMissionName(String missionName) { this.missionName = missionName; }

public int getLaunchYear() { return launchYear; }

public void setLaunchYear(int launchYear) { this.launchYear = launchYear; }

public String getStatus() { return status; }

public void setStatus(String status) { this.status = status; }

}

**2. SpaceMissionRepository.java (Repository Interface)**

package com.spaceagency.missioncontrol.repository;

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

import com.spaceagency.missioncontrol.model.SpaceMission;

public interface SpaceMissionRepository extends JpaRepository<SpaceMission, String> {

}

**3. SpaceMissionService.java (Service Class)**

package com.spaceagency.missioncontrol.service;

import java.util.List;

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

import org.springframework.stereotype.Service;

import com.spaceagency.missioncontrol.model.SpaceMission;

import com.spaceagency.missioncontrol.repository.SpaceMissionRepository;

@Service

public class SpaceMissionService {

@Autowired

private SpaceMissionRepository repository;

public List<SpaceMission> getAllMissions() {

return repository.findAll();

}

}

**4. MissionControlApplication.java (Main Class)**

package com.spaceagency.missioncontrol;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import com.spaceagency.missioncontrol.service.SpaceMissionService;

import com.spaceagency.missioncontrol.model.SpaceMission;

import java.util.List;

@SpringBootApplication

public class MissionControlApplication {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(MissionControlApplication.class, args);

SpaceMissionService service = context.getBean(SpaceMissionService.class);

List<SpaceMission> missions = service.getAllMissions();

for (SpaceMission mission : missions) {

System.out.println(mission.getMissionId() + " - " + mission.getMissionName() + " - " + mission.getLaunchYear() + " - " + mission.getStatus());

}

}

}

**5. application.properties (Basic DB Config)**

spring.datasource.url=jdbc:mysql://localhost:3306/missiondb

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=validate

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

**Output:**

MSN001 - Voyager 1 - 1977 - Active

MSN002 - Apollo 11 - 1969 - Completed

**Exercise: Difference Between JPA, Hibernate, and Spring Data JPA**

**1.JPA (Java Persistence API)**

* JPA is a specification, not a tool or framework.
* It is defined under JSR 338 by Oracle.
* JPA provides a set of rules, interfaces, and annotations to perform Object-Relational Mapping (ORM).
* JPA itself does not contain any implementation.
* It allows Java objects (entities) to be mapped to database tables.

**Examples of JPA Interfaces:**EntityManager, Entity, Table, Id, etc.

Important: To use JPA, we need an implementation like Hibernate**.**

**2.Hibernate**

* Hibernate is the most popular implementation of the JPA specification.
* It is an ORM framework that provides functionalities like:  
  ✔ Automatic table mapping  
  ✔ Writing queries using HQL (Hibernate Query Language)  
  ✔ Handling relationships like One-to-Many, Many-to-One  
  ✔ Caching and lazy loading
* Hibernate provides more advanced features than basic JPA.

**Simply put:  
JPA = Guidelines,  
Hibernate = One of the tools that follows those guidelines.**

**3.Spring Data JPA**

* Spring Data JPA is a part of the Spring Data project.
* It sits on top of Hibernate (or any JPA provider).
* Main goal: Reduce boilerplate code for repository operations.
* With Spring Data JPA, we can create a repository by just writing an interface extending JpaRepository or CrudRepository.
* No need to write SQL or HQL manually for basic CRUD operations.

**Example Flow**

I write an Entity class (JPA concept)  
 Hibernate handles the ORM mapping  
 Spring Data JPA gives ready-made repository interfaces to fetch/save data

**Real-World Example**

**In my project:**

* I used @Entity (from JPA) to map my SpaceMission class
* Hibernate internally manages SQL generation
* Spring Data JPA provides SpaceMissionRepository interface with ready-made methods like .findAll() or .save()