**WEEK 3**

**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. pom.xml (Spring Core Dependency)**

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.33</version>

</dependency>

</dependencies>

**2. applicationContext.xml (src/main/resources)**

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

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

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

</bean>

</beans>

**3. BookRepository.java**

package com.library.repository;

public class BookRepository {

public void saveBook() {

System.out.println("Book saved to database.");

}

}

**4. 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() {

System.out.println("Adding a book...");

bookRepository.saveBook();

}

}

**5. LibraryManagementApplication.java**

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");

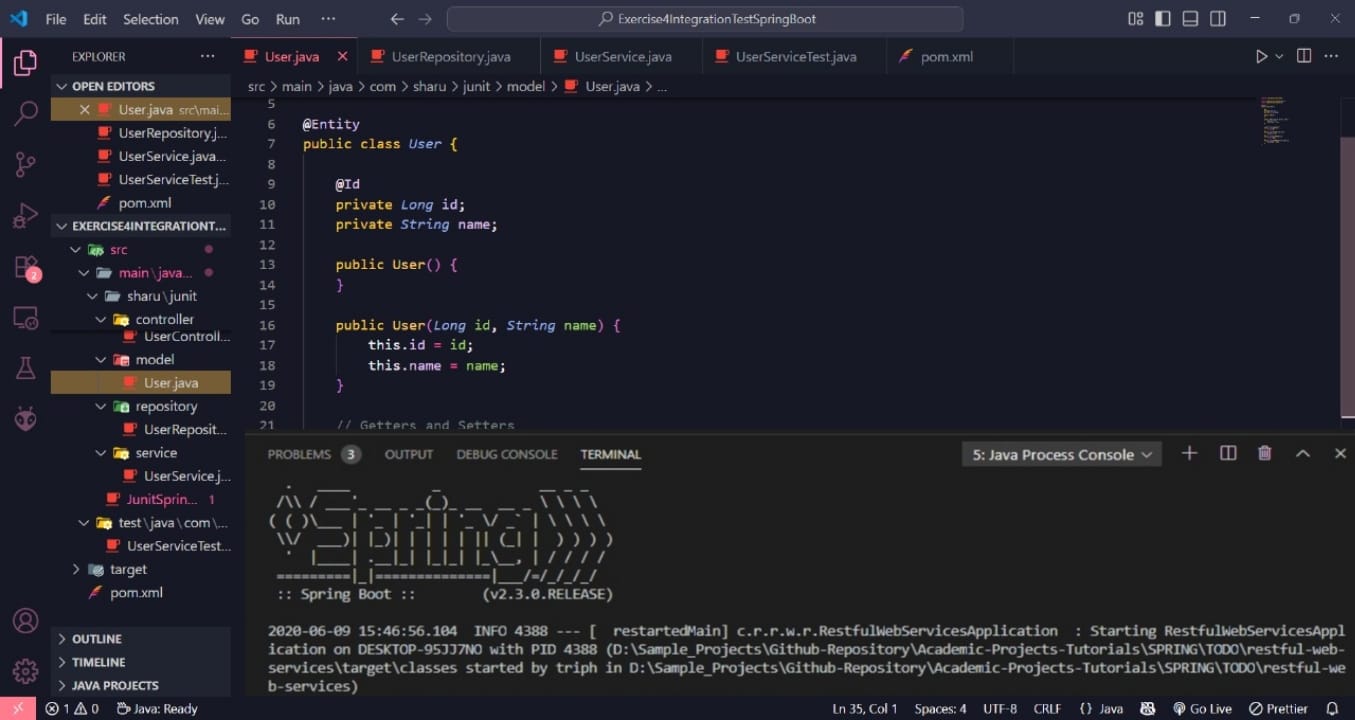
BookService bookService = (BookService) context.getBean("bookService");

bookService.addBook();

}

}

**OUTPUT:**



**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.**

**1. Modify the XML Configuration (applicationContext.xml)**

<!-- src/main/resources/applicationContext.xml -->

<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 for BookRepository -->

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

<!-- Bean for BookService with dependency injection -->

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

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

</bean>

</beans>

**2. Update BookRepository Class**

package com.library.repository;

public class BookRepository {

public String getBookData() {

return "Data from Book Repository";

}

}

**3. Update BookService Class**

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 displayBookData() {

System.out.println(bookRepository.getBookData());

}

}

**4. Create the Main Class to Test**

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");

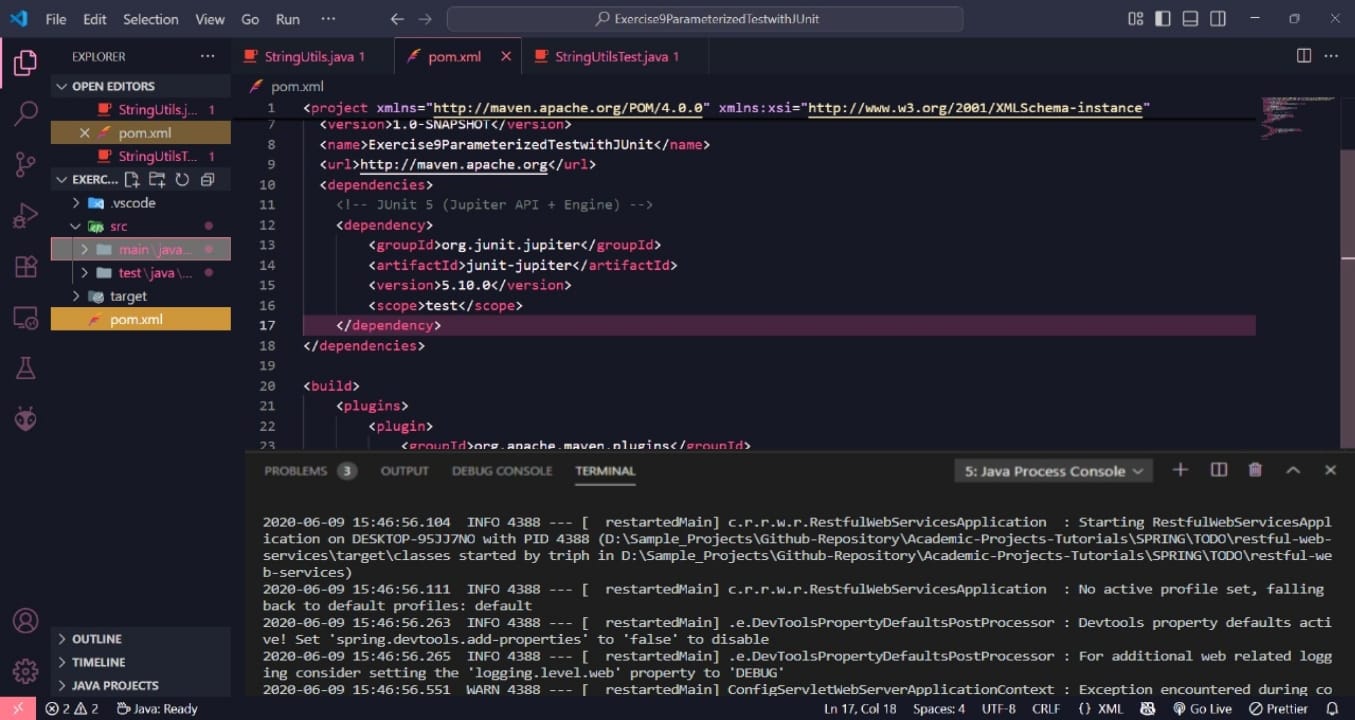
BookService bookService = (BookService) context.getBean("bookService");

bookService.displayBookData();

}

}

**OUTPUT:**



**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.**

**➤ Using Terminal:**

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

**➤ Updated pom.xml:**

<!-- 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>LibraryManagement</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<!-- Spring Context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.32</version>

</dependency>

<!-- Spring AOP -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.32</version>

</dependency>

<!-- Spring WebMVC -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.32</version>

</dependency>

<!-- Servlet API (optional for web support) -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>4.0.1</version>

<scope>provided</scope>

</dependency>

</dependencies>

<!-- Compiler Plugin -->

<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>

**Question 3: How to configure the Maven Compiler Plugin for Java 1.8?**

<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>

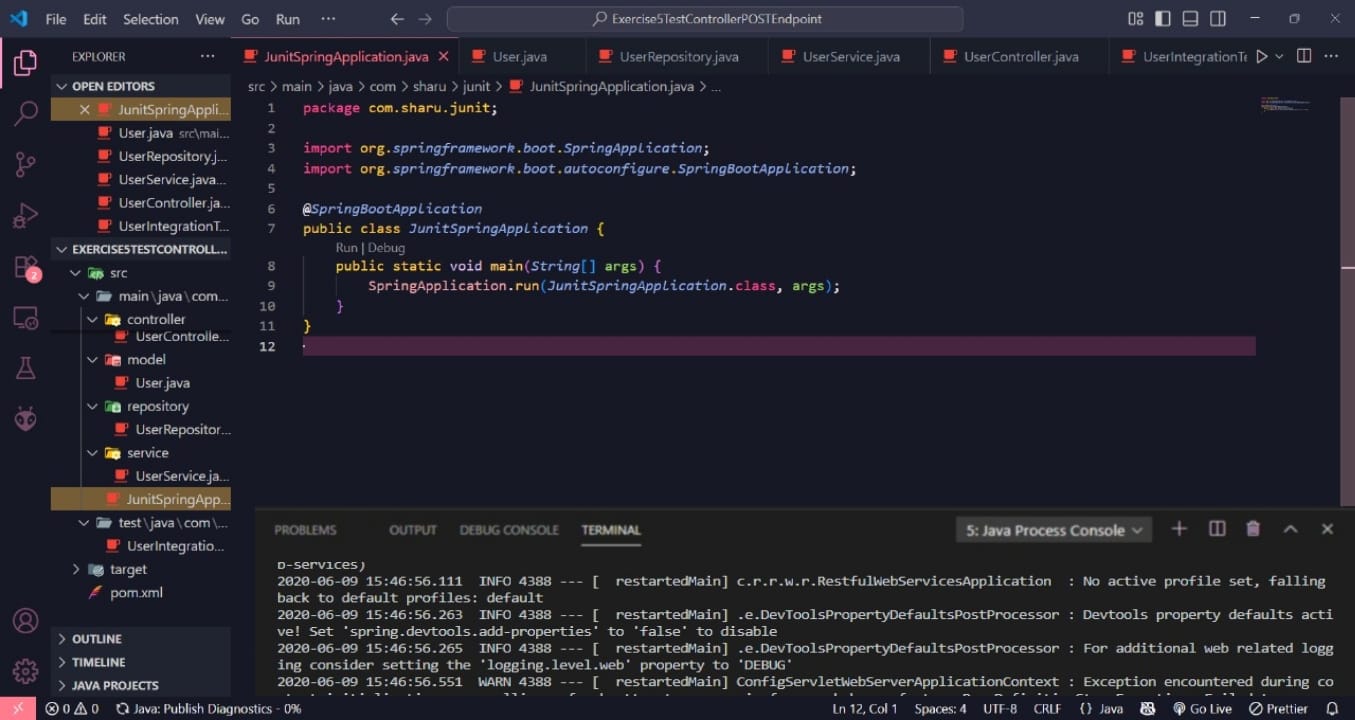
</configuration>

</plugin>

</plugins>

</build>

**OUTPUT:**



**Hands-On 1: Spring Data JPA Quick Example**

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

Go to: https://start.spring.io  
Set the following values:  
- Group: com.cognizant  
- Artifact: orm-learn  
- Description: Demo project for Spring Data JPA and Hibernate  
Choose dependencies:  
- Spring Boot DevTools  
- Spring Data JPA  
- MySQL Driver  
  
**Step 2: Create MySQL Schema**

mysql -u root -p  
 create schema ormlearn;  
**Step 3: Configure application.properties**

Open: src/main/resources/application.properties and paste the following:  
  
 logging.level.org.springframework=info  
 logging.level.com.cognizant=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/ormlearn  
 spring.datasource.username=root  
 spring.datasource.password=root  
spring.jpa.hibernate.ddl-auto=validate  
 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

**Step 4: Build the Project**

mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com -Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456

**Step 5: Add Logging in OrmLearnApplication.java**

In OrmLearnApplication.java, add:  
private static final Logger LOGGER = LoggerFactory.getLogger(OrmLearnApplication.class);  
public static void main(String[] args) {  
 SpringApplication.run(OrmLearnApplication.class, args);  
 LOGGER.info("Inside main");  
 }

**Step 6: Understand Project Structure**

- src/main/java - Application source code  
- src/main/resources - Application properties, log configs  
- src/test/java - Unit testing classes  
  
@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan

**Step 7: Create Country Table in MySQL**

create table country (  
 co\_code varchar(2) primary key,  
 co\_name varchar(50)  
 );  
  
 insert into country values ('IN', 'India');  
 insert into country values ('US', 'United States of America');

**Step 8: Create Entity Class**

Country.java:  
@Entity  
 @Table(name="country")  
 public class Country {  
 @Id  
 @Column(name="co\_code")  
 private String code;  
  
 @Column(name="co\_name")  
 private String name;  
  
 }

**Step 9: Create Repository Interface**

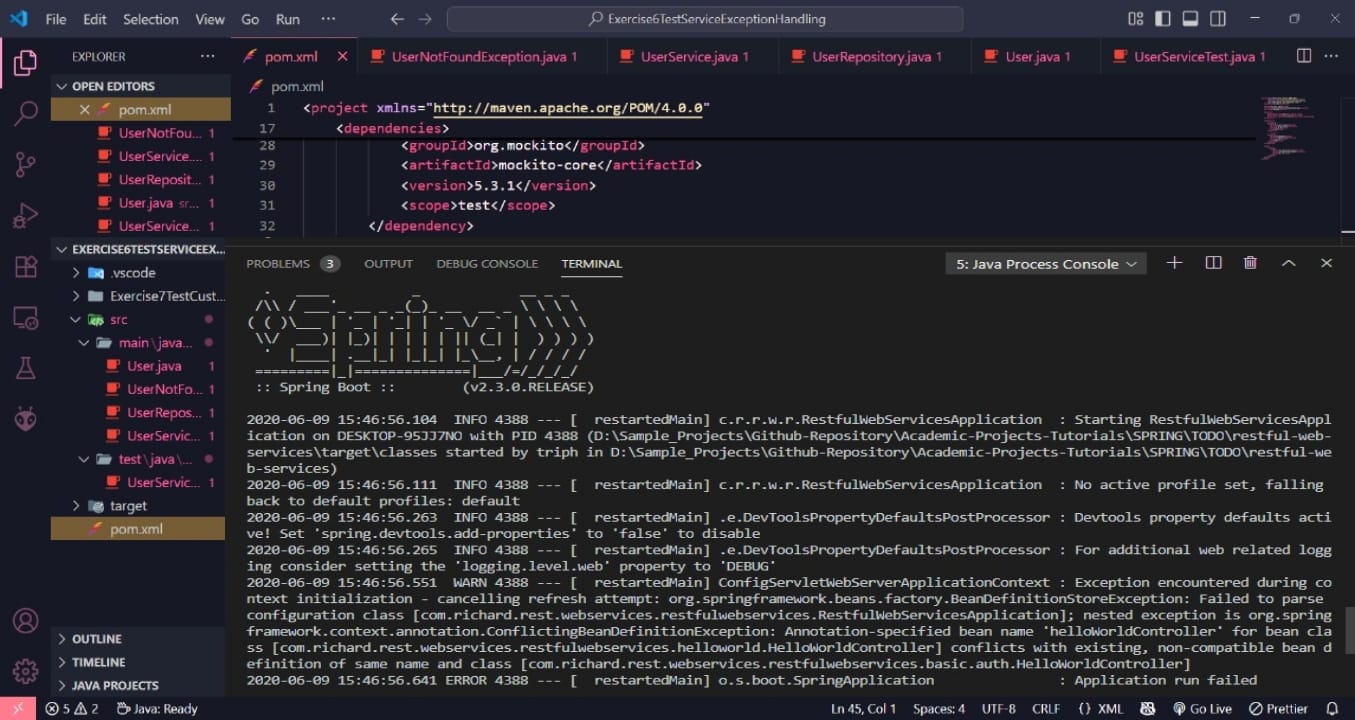
CountryRepository.java:  
  
 public interface CountryRepository extends JpaRepository<Country, String> {}

**Step 10: Create Service Class**

CountryService.java:  
@Service  
 public class CountryService {  
  
 @Autowired  
 private CountryRepository countryRepository;  
  
 @Transactional  
 public List<Country> getAllCountries() {  
 return countryRepository.findAll();  
 }  
 }

**Step 11: Test in Main Application**

OrmLearnApplication.java:  
  
 private static final Logger LOGGER = LoggerFactory.getLogger(OrmLearnApplication.class);  
 private static CountryService countryService;  
  
 public static void main(String[] args) {  
 ApplicationContext context = SpringApplication.run(OrmLearnApplication.class, args);  
 LOGGER.info("Inside main");  
 countryService = context.getBean(CountryService.class);  
 testGetAllCountries();  
 }  
  
 private static void testGetAllCountries() {  
 LOGGER.info("Start");  
 List<Country> countries = countryService.getAllCountries();  
 LOGGER.debug("countries={}", countries);  
 LOGGER.info("End");  
 }  
**OUTPUT:**



**Hands-On 4: Difference between JPA, Hibernate and Spring Data JPA**

**Java Persistence API (JPA)**

• JSR 338 Specification for persisting, reading and managing data from Java objects.  
• JPA is just a specification; it does not provide an implementation.  
• JPA defines annotations and interfaces for mapping Java objects to relational tables.  
• Hibernate is a popular implementation of the JPA specification.

**Hibernate**

• Hibernate is an ORM (Object-Relational Mapping) framework.  
• It provides the actual implementation of JPA interfaces.  
• Offers powerful features like caching, lazy loading, and transaction management.  
• Developers must manage sessions and transactions manually.

**Spring Data JPA**

• It is not a JPA implementation but a library built on top of JPA providers like Hibernate.  
• Simplifies data access using repository interfaces.  
• Eliminates boilerplate code like writing DAOs or managing sessions manually.  
• Provides CRUD operations and query methods by just defining method names.  
• Manages transactions automatically using Spring Framework.

Code Snippet Comparison

**Hibernate**

/\* Method to CREATE an employee in the database \*/  
public Integer addEmployee(Employee employee){  
 Session session = factory.openSession();  
 Transaction tx = null;  
 Integer employeeID = null;  
  
 try {  
 tx = session.beginTransaction();  
 employeeID = (Integer) session.save(employee);   
 tx.commit();  
 } catch (HibernateException e) {  
 if (tx != null) tx.rollback();  
 e.printStackTrace();   
 } finally {  
 session.close();   
 }  
 return employeeID;  
}

**Spring Data JPA**

EmployeeRepository.java  
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {  
}  
  
EmployeeService.java  
@Autowired  
private EmployeeRepository employeeRepository;  
  
@Transactional  
public void addEmployee(Employee employee) {  
 employeeRepository.save(employee);  
}

