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**.
    - Create a package **com.library.repository** and add a class **BookRepository**.
4. **Run the Application:**
    - Create a main class to load the Spring context and test the configuration.

Process:

Step 1: Set Up a Spring Project

1.1 Create a Maven Project

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

1.2 Update pom.xml to Add Spring Core Dependencies

```
<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>5.3.33</version>
    </dependency>
  </dependencies>
```

```
    </project>
```

Step 2: Configure the Application Context

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

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

Step 3: Define Service and Repository Classes

3.1 Create BookRepository class

```
package com.library.repository;
public class BookRepository {
   public String getBook() {
      return "Effective Java by Joshua Bloch";
   }
}
```

3.2 Create 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;
   }
```

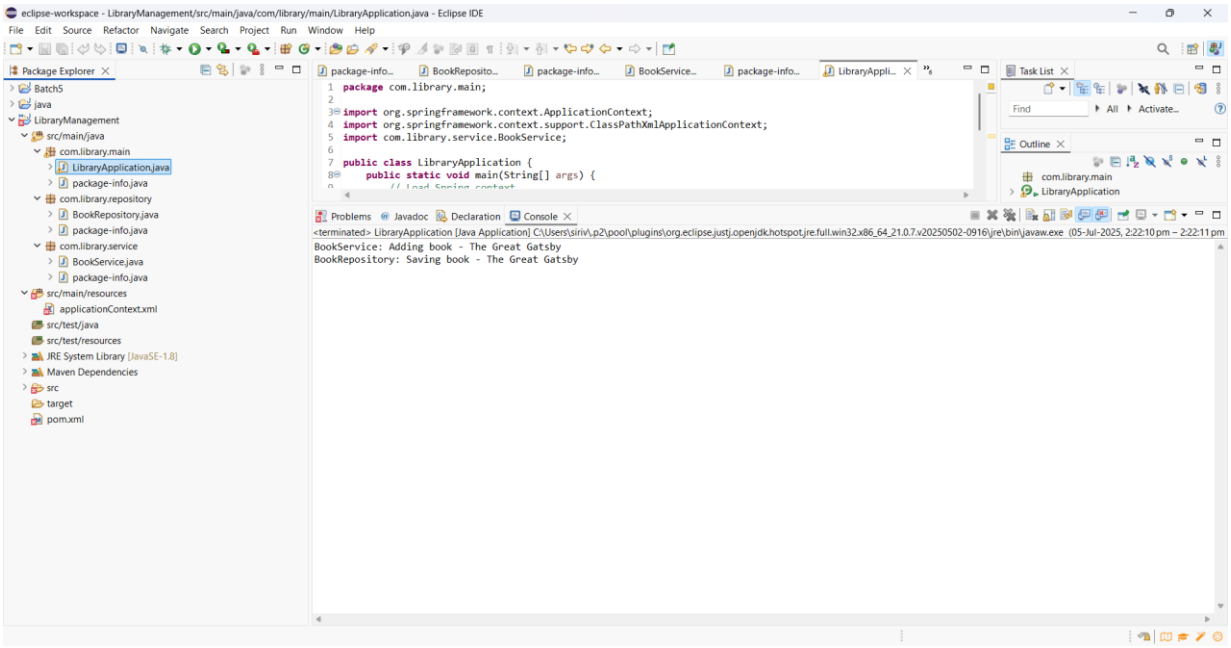
```
 public void displayBook() {
        String book = bookRepository.getBook();
        System.out.println("Book: " + book);
    }
}
```

Step 4: Run the Application

4.1 Create the Main Class

```
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) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
 BookService bookService = (BookService) context.getBean("bookService");
        bookService.displayBook();
    }
}
```

Step 5: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.

Steps:

1. **Modify the XML Configuration:**
    - **Update applicationContext.xml to wire BookRepository into BookService.**
2. **Update the BookService Class:**
    - **Ensure that BookService class has a setter method for BookRepository.**
3. **Test the Configuration:**
    - **Run the LibraryManagementApplication main class to verify the dependency injection.**

Process:

Step 1: Modify the XML Configuration

```
<?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">
```

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

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

</beans>
```

Step 2: Ensure BookService Has a Setter Method

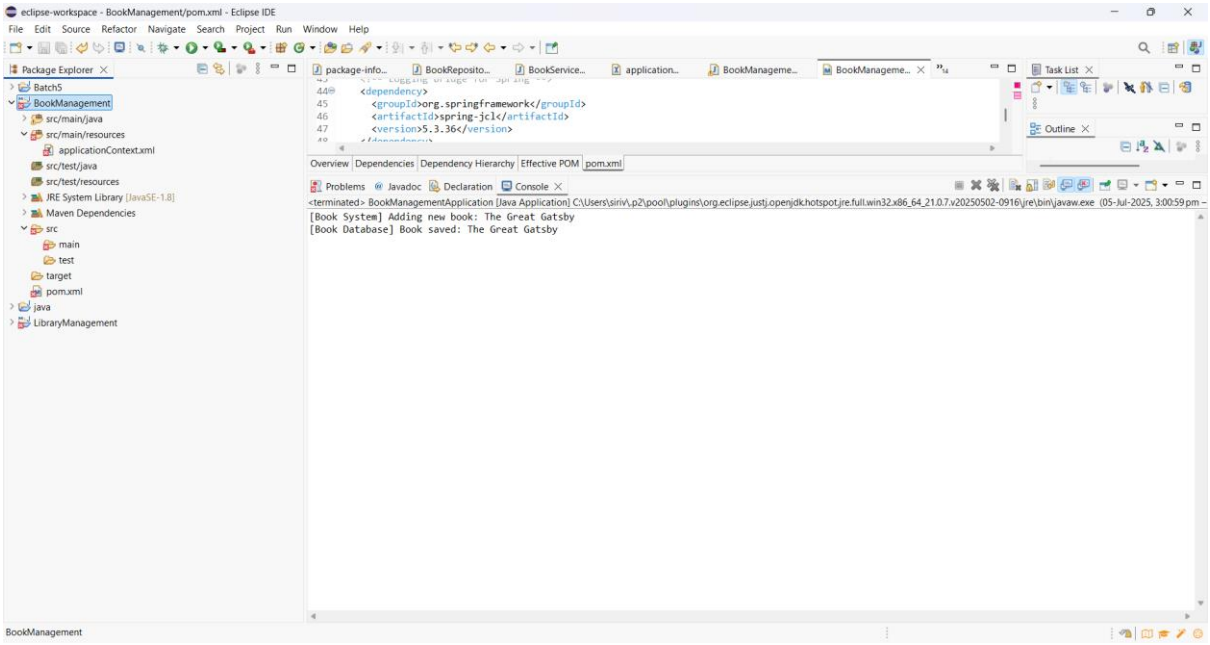
```
package com.library.service;
import com.library.repository.BookRepository;
public class BookService {
    private BookRepository bookRepository;
    public void setBookRepository(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }
public void displayBook() {
        String book = bookRepository.getBook();
        System.out.println("Book: " + book);
    }
}
```

Step 3: Test the Configuration

```
package com.library;
import com.library.service.BookService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class BookManagementApplication {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        BookService bookService = (BookService) context.getBean("bookService");
```

```
        bookService.displayBook();
    }
}
```

Step 4:Out put



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

Process:

Step 1: Create a New Maven Project

- New Project > Maven >

- GroupId: com.store
- ArtifactId: StoreManagement

Step 2: Add Spring Dependencies

```
<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.store</groupId>
    <artifactId>StoreManagement</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <!-- Spring Context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.33</version>
        </dependency>
 <!-- Spring AOP -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>5.3.33</version>
        </dependency>

        <!-- Spring Web MVC -->
        <dependency>
```

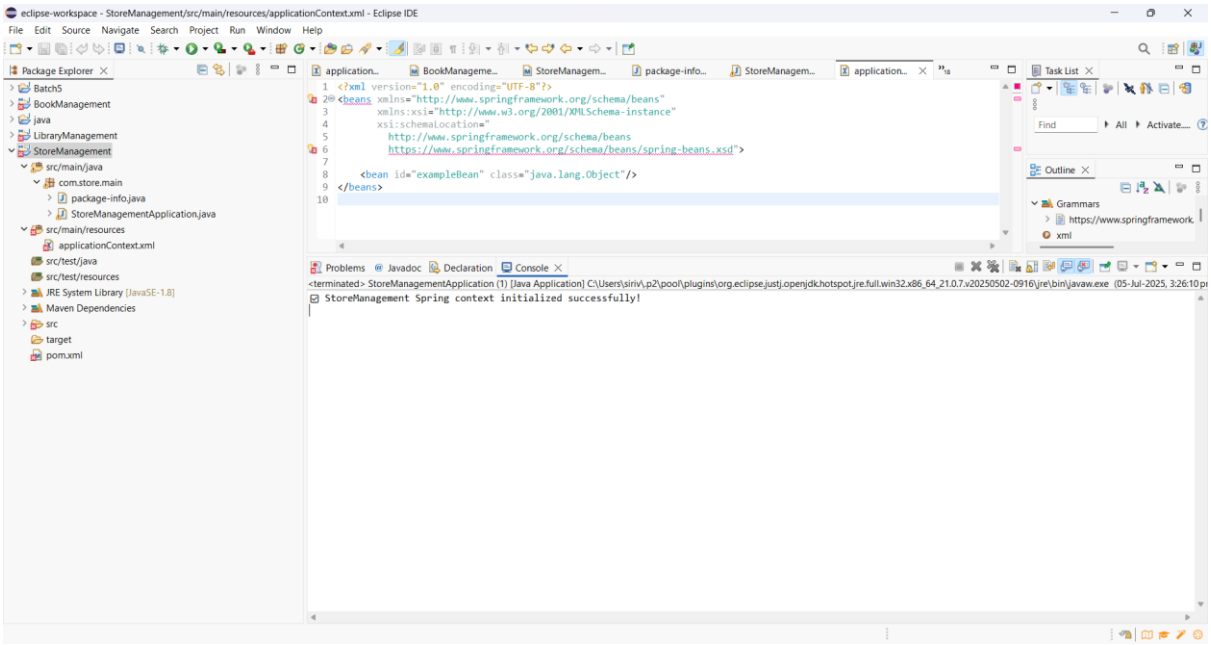
```
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.3.33</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- Maven Compiler Plugin -->
            <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>
```

Step 3: Main class

```
package com.store.main;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class StoreManagementApplication {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        System.out.println("✅ StoreManagement Spring context initialized successfully!");
    }
}
```

Step 4:Out put



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.**
2. **Update the BookService Class:**
    - **Ensure that the BookService class has a setter method for BookRepository.**
3. **Run the Application:**
    - **Create a main class to load the Spring context and test the configuration.**

Process:

Step 1: Create Spring Configuration File

<?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">

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

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

</beans>
```

Step 2: Update the ProductService Class

```
package com.product.service;
import com.product.repository.ProductRepository;
public class ProductService {
    private ProductRepository productRepository;
    public void setProductRepository(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }
public void performOperation() {
        System.out.println("✓ ProductService: Performing product operation...");
        productRepository.saveProduct();
    }
}
```

Step 3: Create and Run the Main Class

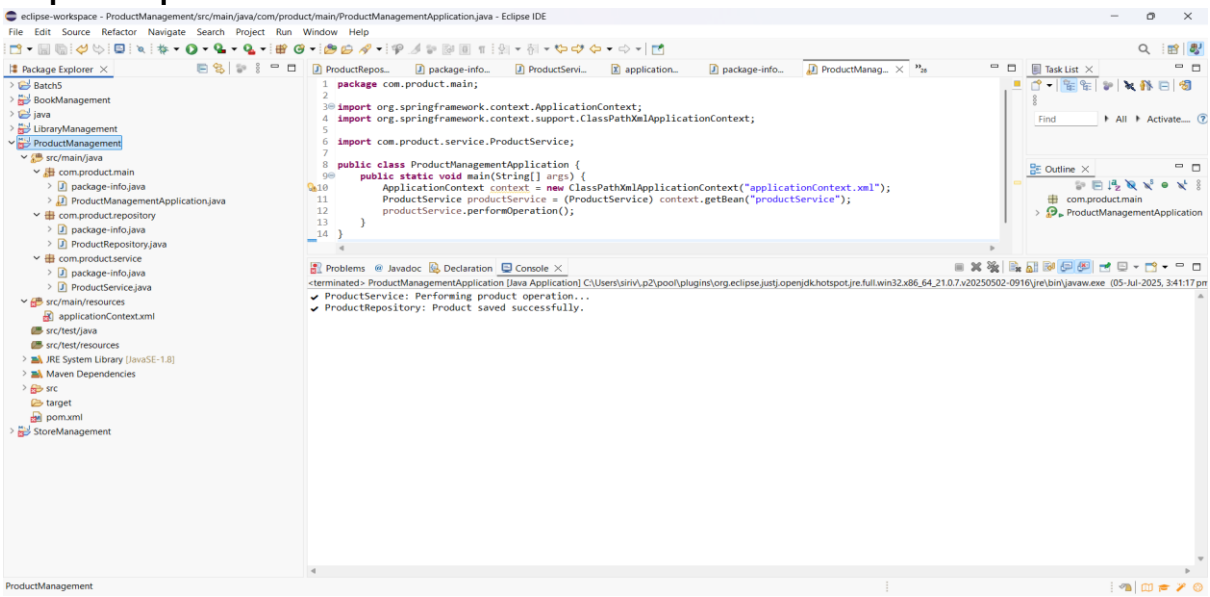
```
package com.product.main;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.product.service.ProductService;
```

```
public class ProductManagementApplication {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        ProductService productService = (ProductService) context.getBean("productService");
        productService.performOperation();
    }
}
```

Step 4: Update the ProductRepository Class

```
package com.product.repository;
public class ProductRepository {
    public void saveProduct() {
        System.out.println("✓ ProductRepository: Product saved successfully.");
    }
}
```

Step 5:Out put



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.**

2. **Configure Setter Injection:**
    - **Ensure that the BookService class has a setter method for BookRepository and configure it in applicationContext.xml.**
3. **Test the Injection:**
    - **Run the LibraryManagementApplication main class to verify both constructor and setter injection.**

Process:

Step 1. InventoryRepository.java

```
package com.inventory.repository;

public class ItemRepository {
    public void saveItem() {
        System.out.println("✓ ItemRepository: Item saved successfully.");
    }
}
```

Step 2. InventoryService.java

```
package com.inventory.service;

import com.inventory.repository.ItemRepository;

public class ItemService {
    private ItemRepository itemRepository;
    private String serviceName;
    public ItemService(String serviceName) {
        this.serviceName = serviceName;
    }
    public void setItemRepository(ItemRepository itemRepository) {
        this.itemRepository = itemRepository;
    }
public void performOperation() {
        System.out.println("✓ ItemService [" + serviceName + "]: Performing inventory operation...");
        itemRepository.saveItem();
    }
```

```
}
```

Step 3. InventoryManagement.java

```
package com.inventory.main;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.inventory.service.ItemService;
public class InventoryManagementApplication {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        ItemService itemService = (ItemService) context.getBean("itemService");
        itemService.performOperation();
    }
}
```

Step 4. applicationContext.xml

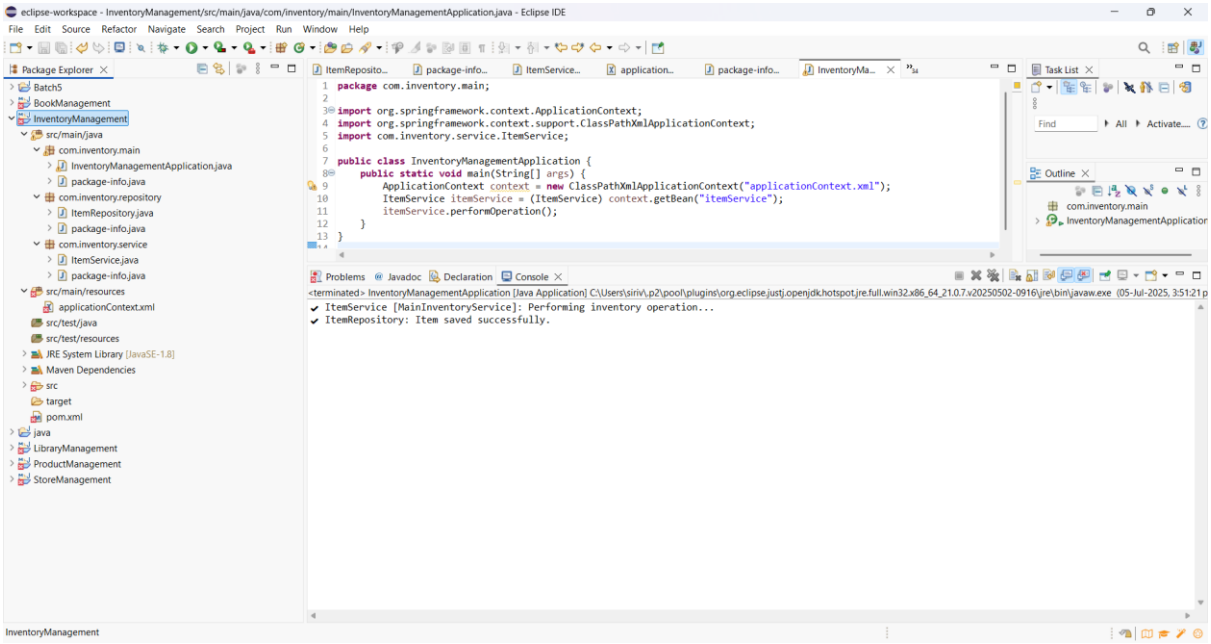
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- Bean for ItemRepository -->
    <bean id="itemRepository" class="com.inventory.repository.ItemRepository"/>

    <!-- Bean for ItemService with constructor and setter injection -->
    <bean id="itemService" class="com.inventory.service.ItemService">
        <!-- Constructor injection -->
        <constructor-arg value="MainInventoryService"/>
        <!-- Setter injection -->
        <property name="itemRepository" ref="itemRepository"/>
    </bean>
```

</beans>

Step 5:Out put



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.**
3. **Create Application Properties:**
    - **Configure database connection properties in application.properties.**
4. **Define Entities and Repositories:**
    - **Create Book entity and BookRepository interface.**
5. **Create a REST Controller:**
    - **Create a BookController class to handle CRUD operations.**
6. **Run the Application:**
    - **Run the Spring Boot application and test the REST endpoints.**

Process:

Step 1: Create a Spring Boot Project

Use Spring Initializr or your IDE to create a project:

- Project name: OrderManagement
- Group: com.store
- Artifact: OrderManagement
- Dependencies:
    - Spring Web
    - Spring Data JPA
    - H2 Database

Step 2: Add Dependencies

```
<dependencies>
    <!-- Spring Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

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

    <!-- H2 In-Memory Database -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
```

```
    <!-- Spring Boot Starter -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
</dependencies>
```

Step 3: Create application.properties

```
spring.datasource.url=jdbc:h2:mem:ordersdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
spring.jpa.hibernate.ddl-auto=update
```

Step 4: Define Entity and Repository

Order.java

```
package com.store.entity;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
@Entity
public class Order {
@Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
private String itemName;
    private int quantity;
    private double price;
    public Long getId() {
        return id;
    }
```

```
 public void setId(Long id) {
        this.id = id;
    }
 public String getItemName() {
        return itemName;
    }
 public void setItemName(String itemName) {
        this.itemName = itemName;
    }
 public int getQuantity() {
        return quantity;
    }
public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
 public double getPrice() {
        return price;
    }
 public void setPrice(double price) {
        this.price = price;
    }
}
```

OrderRepository.java

```
package com.store.repository;
import com.store.entity.Order;
import org.springframework.data.jpa.repository.JpaRepository;
public interface OrderRepository extends JpaRepository<Order, Long> {
}
```

Step 5: Create REST Controller

```
package com.store.controller;

import com.store.entity.Order;
```

```
import com.store.repository.OrderRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    private OrderRepository orderRepository;

    @GetMapping
    public List<Order> getAllOrders() {
        return orderRepository.findAll();
    }

    @GetMapping("/{id}")
    public Optional<Order> getOrderById(@PathVariable Long id) {
        return orderRepository.findById(id);
    }

    @PostMapping
    public Order createOrder(@RequestBody Order order) {
        return orderRepository.save(order);
    }

    @PutMapping("/{id}")
    public Order updateOrder(@PathVariable Long id, @RequestBody Order updatedOrder) {
        return orderRepository.findById(id).map(order -> {
```

```
            order.setItemName(updatedOrder.getItemName());
            order.setQuantity(updatedOrder.getQuantity());
            order.setPrice(updatedOrder.getPrice());
            return orderRepository.save(order);
        }).orElseGet(() -> {
            updatedOrder.setId(id);
            return orderRepository.save(updatedOrder);
        });
    }

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

Step 6: Main Class – OrderManagementApplication.java

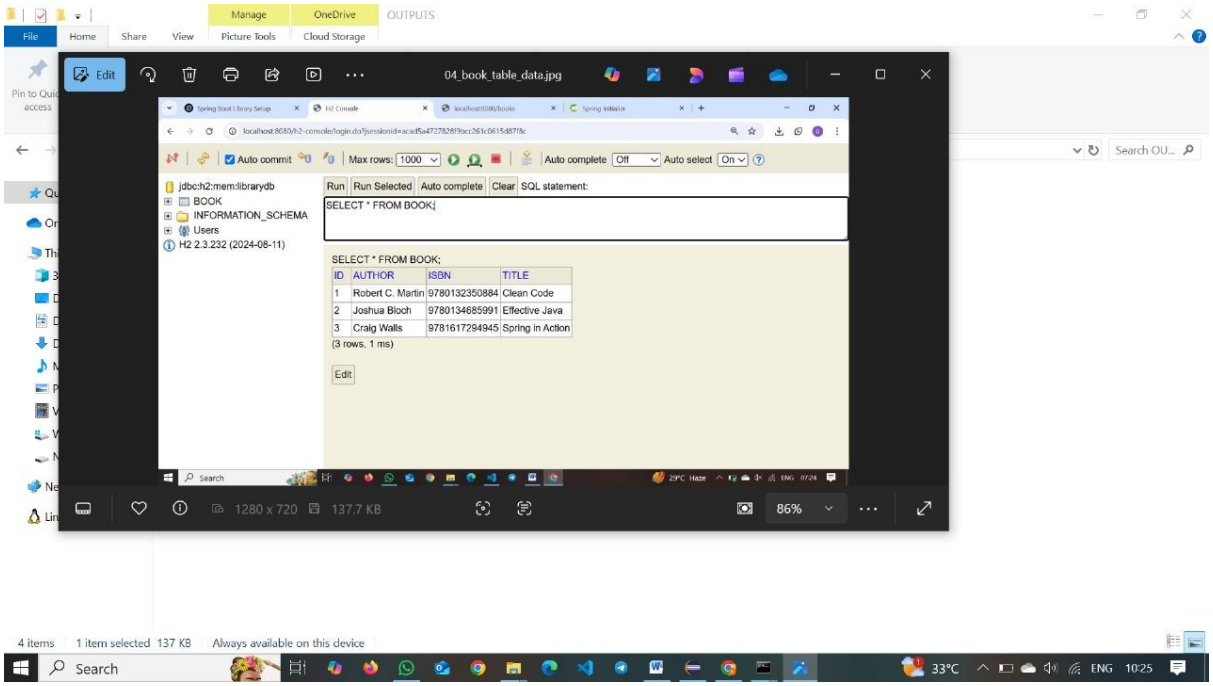
```
package com.store;

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

@SpringBootApplication
public class OrderManagementApplication {

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

Step 7: Out put


04_book_table_data.jpg
jdbc:h2:mem:librarydb
BOOK
INFORMATION_SCHEMA
Users
H2 2.3.232 (2024-08-11)
Run
Run Selected
Auto complete
Clear
SQL statement:
SELECT * FROM BOOK;
SELECT * FROM BOOK;
ID AUTHOR ISBN TITLE
1 Robert C. Martin 9780132350884 Clean Code
2 Joshua Bloch 9780134685991 Effective Java
3 Craig Walls 9781617294945 Spring in Action
(3 rows, 1 ms)
Edit
1280 x 720
137.7 KB
86%