**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.

To set up a basic Spring application for managing a library, follow the steps outlined below:

Step 1: Set Up a Spring Project

1.Create a Maven Project:

- Use your IDE (like IntelliJ IDEA or Eclipse) to create a new Maven project named `LibraryManagement`.

2. Add Spring Core Dependencies:

- Open the `pom.xml` file and add the following Spring dependencies:

```xml

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.20</version> <!-- Use the latest stable version -->

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>5.3.20</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.32</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.7.32</version>

</dependency>

</dependencies>

```

Step 2: Configure the Application Context

1. Create an XML Configuration File:

- Create a file named `applicationContext.xml` in the `src/main/resources` directory.

2. Define Beans for BookService and BookRepository:

- Add the following XML configuration to `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">

<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

1. Create a Package for Service:

- Create a package named `com.library.service` and add the `BookService` class:

```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 manageBooks() {

// Logic to manage books

System.out.println("Managing books...");

}

}

2. Create a Package for Repository:

- Create a package named `com.library.repository` and add the `BookRepository` class:

```java

package com.library.repository;

public class BookRepository {

// Logic to interact with the database

public void saveBook() {

System.out.println("Saving book...");

}

}

Step 4: Run the Application

1. Create a Main Class:

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

```java

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.manageBooks();

}

}

**xercise 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.

To implement dependency injection in your library management application using Spring's Inversion of Control (IoC) and Dependency Injection (DI), follow the steps outlined below:

Step 1: Modify the XML Configuration

1. Update `applicationContext.xml:

- Ensure that the `BookRepository` is wired into the `BookService` using the setter method. The XML configuration should already reflect this, but here’s a reminder of how it should look:

```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">

<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 BookService Class

1. Ensure that `BookService` has a Setter Method:

- The `BookService` class should already have a setter method for `BookRepository`. Here’s the updated class for reference:

```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 manageBooks() {

// Logic to manage books

System.out.println("Managing books...");

bookRepository.saveBook(); // Example usage of the repository

}

}

```

Step 3: Test the Configuration

1.Run the `LibraryManagementApplication` Main Class:

- Ensure that your main class is set up to load the Spring context and test the dependency injection. Here’s the main class for reference:

```java

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.manageBooks();

}

}

```

Expected Output

- When you run the `LibraryManagementApplication`, you should see the following output:

```

Managing books...

Saving book...

```

This output indicates that the `BookService` is successfully managing books and is able to call the `saveBook` method from the `BookRepository`, demonstrating that dependency injection has been correctly implemented.

**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.

To create and configure a new Maven project for the library management application with the necessary Spring dependencies, follow the steps outlined below:

Step 1: Create a New Maven Project

1. Create a New Maven Project:

- Use your IDE (like IntelliJ IDEA or Eclipse) to create a new Maven project named `LibraryManagement`.

- If you are using the command line, you can create a new Maven project using the following command:

```bash

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

```

Step 2: Add Spring Dependencies in `pom.xml`

1. Open the `pom.xml` File:

- Navigate to the `pom.xml` file in your project.

2. Include Dependencies for Spring Context, Spring AOP, and Spring WebMVC:

- Add the following dependencies inside the `<dependencies>` tag:

```xml

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.20</version> <!-- Use the latest stable version -->

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.20</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.20</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

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

<version>4.0.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.32</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.7.32</version>

</dependency>

</dependencies>

```

Step 3: Configure Maven Plugins

1. Configure the Maven Compiler Plugin:

- Add the Maven Compiler Plugin configuration to specify the Java version. This should be added inside the `<build>` tag in your `pom.xml`:

```xml

<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>

```

Complete `pom.xml` Example

Here’s how your complete `pom.xml` file should look after making the above changes:

```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>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.20</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.20</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.20</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

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

<version>4.0.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.32</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.7.32</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>

```

**Implement services for managing Country**   
  
An application requires for features to be implemented with regards to country. These features needs to be supported by implementing them as service using Spring Data JPA.

* Find a country based on country code
* Add new country
* Update country
* Delete country
* Find list of countries matching a partial country name

@Entity

public class Country {

@Id

@Column(length = 3)

private String code; // ISO country code

@Column(nullable = false)

private String name;

// Constructors

public Country() {}

public Country(String code, String name) {

this.code = code;

this.name = name;

}

// Getters & Setters

}

public interface CountryRepository extends JpaRepository<Country, String> {

List<Country> findByNameContainingIgnoreCase(String partialName);

}

@Service

public class CountryService {

@Autowired

private CountryRepository countryRepository;

public Country findByCode(String code) {

return countryRepository.findById(code)

.orElseThrow(() -> new NoSuchElementException("Country not found"));

}

public Country addCountry(Country country) {

if (countryRepository.existsById(country.getCode())) {

throw new IllegalArgumentException("Country already exists");

}

return countryRepository.save(country);

}

public Country updateCountry(String code, Country updatedCountry) {

Country existing = findByCode(code);

existing.setName(updatedCountry.getName());

return countryRepository.save(existing);

}

public void deleteCountry(String code) {

if (!countryRepository.existsById(code)) {

throw new NoSuchElementException("Country not found");

}

countryRepository.deleteById(code);

}

public List<Country> searchByPartialName(String partialName) {

return countryRepository.findByNameContainingIgnoreCase(partialName);

}

}

@RestController

@RequestMapping("/api/countries")

public class CountryController {

@Autowired

private CountryService countryService;

@GetMapping("/{code}")

public ResponseEntity<Country> getByCode(@PathVariable String code) {

return ResponseEntity.ok(countryService.findByCode(code));

}

@PostMapping

public ResponseEntity<Country> add(@RequestBody Country country) {

return new ResponseEntity<>(countryService.addCountry(country), HttpStatus.CREATED);

}

@PutMapping("/{code}")

public ResponseEntity<Country> update(@PathVariable String code, @RequestBody Country updated) {

return ResponseEntity.ok(countryService.updateCountry(code, updated));

}

@DeleteMapping("/{code}")

public ResponseEntity<Void> delete(@PathVariable String code) {

countryService.deleteCountry(code);

return ResponseEntity.noContent().build();

}

@GetMapping("/search")

public ResponseEntity<List<Country>> search(@RequestParam String name) {

return ResponseEntity.ok(countryService.searchByPartialName(name));

}

}

**Find a country based on country code** 

* Create new exception class CountryNotFoundException in com.cognizant.spring-learn.service.exception
* Create new method findCountryByCode() in CountryService with @Transactional annotation
* In findCountryByCode() method, perform the following steps:
  + Method signature

@Transactional

public Country findCountryByCode(String countryCode) throws CountryNotFoundException

* Get the country based on findById() built in method

Optional<Country> result = countryRepository.findById(countryCode);

* From the result, check if a country is found. If not found, throw CountryNotFoundException

if (!result.isPresent())

* Use get() method to return the country fetched.

Country country = result.get();

* Include new test method in OrmLearnApplication to find a country based on country code and compare the country name to check if it is valid.

    private static void getAllCountriesTest() {

        LOGGER.info("Start");

        Country country = countryService.findCountryByCode("IN");

  LOGGER.debug("Country:{}", country);

        LOGGER.info("End");

    }

* Invoke the above method in main() method and test it.

**NOTE:** SME to explain the importance of @Transactional annotation. Spring takes care of creating the Hibernate session and manages the transactionality when executing the service method.

To implement the functionality of finding a country based on its country code in your Spring application, follow the steps outlined below. This includes creating a custom exception, implementing the service method, and testing the functionality.

Step 1: Create the `CountryNotFoundException` Class

1. Create a New Exception Class:

- Create a new class named `CountryNotFoundException` in the package `com.cognizant.spring-learn.service.exception`.

```java

package com.cognizant.spring-learn.service.exception;

public class CountryNotFoundException extends Exception {

public CountryNotFoundException(String message) {

super(message);

}

}

Step 2: Implement the `findCountryByCode` Method in `CountryService`

1. Update the `CountryService` Class:

- Add the `findCountryByCode` method with the `@Transactional` annotation.

```java

package com.cognizant.spring-learn.service;

import com.cognizant.spring-learn.service.exception.CountryNotFoundException;

import com.cognizant.spring-learn.model.Country;

import com.cognizant.spring-learn.repository.CountryRepository;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.util.Optional;

@Service

public class CountryService {

@Autowired

private CountryRepository countryRepository;

@Transactional

public Country findCountryByCode(String countryCode) throws CountryNotFoundException {

Optional<Country> result = countryRepository.findById(countryCode);

if (!result.isPresent()) {

throw new CountryNotFoundException("Country not found for code: " + countryCode);

}

return result.get();

}

}

```

Step 3: Create a Test Method in `OrmLearnApplication`

1. \*\*Update the `OrmLearnApplication` Class\*\*:

- Add a test method to find a country based on its code and log the result.

```java

package com.cognizant.spring-learn;

import com.cognizant.spring-learn.model.Country;

import com.cognizant.spring-learn.service.CountryService;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class OrmLearnApplication implements CommandLineRunner {

private static final Logger LOGGER = LoggerFactory.getLogger(OrmLearnApplication.class);

@Autowired

private CountryService countryService;

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

getAllCountriesTest();

}

private void getAllCountriesTest() {

LOGGER.info("Start");

try {

Country country = countryService.findCountryByCode("IN");

LOGGER.debug("Country: {}", country);

} catch (CountryNotFoundException e) {

LOGGER.error("Error: {}", e.getMessage());

}

LOGGER.info("End");

}

}

```

Step 4: Explanation of the `@Transactional` Annotation

Importance of `@Transactional:

- The `@Transactional` annotation is crucial in Spring applications as it manages the transaction boundaries for the method it annotates. When a method is marked with `@Transactional`, Spring takes care of creating a Hibernate session and managing the transactionality when executing the service method.

- If the method completes successfully, the transaction is committed, ensuring that all changes are saved to the database. If an exception occurs, the transaction is rolled back, preventing any partial updates to the database. This ensures data integrity and consistency.

**Add a new country** 

* Create new method in CountryService.

@Transactional

public void addCountry(Country country)

* Invoke save() method of repository to get the country added.

countryRepository.save(country)

* Include new testAddCountry() method in OrmLearnApplication. Perform steps below:
  + Create new instance of country with a new code and name
  + Call countryService.addCountry() passing the country created in the previous step.
  + Invoke countryService.findCountryByCode() passing the same code used when adding a new country
  + Check in the database if the country is added

To implement the functionality for adding a new country in your Spring application, follow the steps outlined below. This includes creating a method in the `CountryService` to add a country, and testing this functionality in the `OrmLearnApplication`.

Step 1: Create the `addCountry` Method in `CountryService`

1. Update the `CountryService` Class:

- Add the `addCountry` method with the `@Transactional` annotation.

```java

package com.cognizant.spring-learn.service;

import com.cognizant.spring-learn.model.Country;

import com.cognizant.spring-learn.repository.CountryRepository;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class CountryService {

@Autowired

private CountryRepository countryRepository;

@Transactional

public void addCountry(Country country) {

countryRepository.save(country);

}

// Existing method

@Transactional

public Country findCountryByCode(String countryCode) throws CountryNotFoundException {

Optional<Country> result = countryRepository.findById(countryCode);

if (!result.isPresent()) {

throw new CountryNotFoundException("Country not found for code: " + countryCode);

}

return result.get();

}

}

```

Step 2: Include the `testAddCountry` Method in `OrmLearnApplication`

1. Update the `OrmLearnApplication` Class:

- Add a new method `testAddCountry` to create a new country, add it, and verify that it has been added.

```java

package com.cognizant.spring-learn;

import com.cognizant.spring-learn.model.Country;

import com.cognizant.spring-learn.service.CountryService;

import com.cognizant.spring-learn.service.exception.CountryNotFoundException;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class OrmLearnApplication implements CommandLineRunner {

private static final Logger LOGGER = LoggerFactory.getLogger(OrmLearnApplication.class);

@Autowired

private CountryService countryService;

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

testAddCountry();

}

private void testAddCountry() {

LOGGER.info("Start adding country");

try {

// Create a new instance of Country

Country newCountry = new Country();

newCountry.setCode("US"); // Example country code

newCountry.setName("United States"); // Example country name

// Add the new country

countryService.addCountry(newCountry);

LOGGER.info("Country added: {}", newCountry);

// Verify that the country has been added

Country addedCountry = countryService.findCountryByCode("US");

LOGGER.debug("Added Country: {}", addedCountry);

} catch (CountryNotFoundException e) {

LOGGER.error("Error: {}", e.getMessage());

} catch (Exception e) {

LOGGER.error("An error occurred: {}", e.getMessage());

}

LOGGER.info("End adding country");

}

}

```

Step 3: Verify the Database

- After running the application, you should check your database to ensure that the new country has been added successfully. You can do this by querying the `Country` table to see if the entry for the country code "US" exists.

**Objectives**

* Demonstrate implementation of Query Methods feature of Spring Data JPA
  + Query Methods - Search by containing text, sorting, filter with starting text, fetch between dates, greater than or lesser than, top
    - Query methods - https://docs.spring.io/spring-data/jpa/docs/2.2.0.RELEASE/reference/html/#jpa.query-methods.query-creation
* Demonstrate implementation of O/R Mapping
  + @ManyToOne, @JoinColumn, @OneToMany, FetchType.EAGER, FetchType.LAZY, @ManyToMany, @JoinTable, mappedBy
    - Relationships reference - https://www.baeldung.com/spring-data-rest-relationships

To demonstrate the implementation of Query Methods in Spring Data JPA and Object-Relational Mapping (O/R Mapping) with various relationships, we will create a simple library management application. This application will include entities for `Book`, `Author`, and `Publisher`, showcasing different relationships and query methods.

Step 1: Define Entities with O/R Mapping

1. Create the `Author` Entity:

```java

package com.cognizant.springlearn.model;

import javax.persistence.\*;

import java.util.Set;

@Entity

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "author", fetch = FetchType.LAZY)

private Set<Book> books;

// Getters and Setters

}

```

2. \*\*Create the `Publisher` Entity\*\*:

```java

package com.cognizant.springlearn.model;

import javax.persistence.\*;

import java.util.Set;

@Entity

public class Publisher {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "publisher", fetch = FetchType.LAZY)

private Set<Book> books;

// Getters and Setters

}

```

3. Create the `Book` Entity:

```java

package com.cognizant.springlearn.model;

import javax.persistence.\*;

import java.util.Date;

@Entity

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

@ManyToOne

@JoinColumn(name = "author\_id")

private Author author;

@ManyToOne

@JoinColumn(name = "publisher\_id")

private Publisher publisher;

private Date publishedDate;

// Getters and Setters

}

```

Step 2: Create Repository Interfaces with Query Methods

1. Create the `BookRepository` Interface:

```java

package com.cognizant.springlearn.repository;

import com.cognizant.springlearn.model.Book;

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

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

import java.util.Date;

import java.util.List;

public interface BookRepository extends JpaRepository<Book, Long> {

List<Book> findByTitleContaining(String title);

List<Book> findByPublishedDateBetween(Date startDate, Date endDate);

List<Book> findByTitleStartingWith(String prefix);

List<Book> findTop5ByOrderByPublishedDateDesc();

}

```

Step 3: Implementing the Service Layer

1.Create the `BookService` Class:

```java

package com.cognizant.springlearn.service;

import com.cognizant.springlearn.model.Book;

import com.cognizant.springlearn.repository.BookRepository;

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

import org.springframework.stereotype.Service;

import java.util.Date;

import java.util.List;

@Service

public class BookService {

@Autowired

private BookRepository bookRepository;

public List<Book> searchBooksByTitle(String title) {

return bookRepository.findByTitleContaining(title);

}

public List<Book> getBooksPublishedBetween(Date startDate, Date endDate) {

return bookRepository.findByPublishedDateBetween(startDate, endDate);

}

public List<Book> getBooksStartingWith(String prefix) {

return bookRepository.findByTitleStartingWith(prefix);

}

public List<Book> getTop5RecentBooks() {

return bookRepository.findTop5ByOrderByPublishedDateDesc();

}

}

```

Step 4: Testing the Query Methods

1. Update the `OrmLearnApplication` Class:

```java

package com.cognizant.springlearn;

import com.cognizant.springlearn.model.Book;

import com.cognizant.springlearn.service.BookService;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.Date;

import java.util.List;

@SpringBootApplication

public class OrmLearnApplication implements CommandLineRunner {

private static final Logger LOGGER = LoggerFactory.getLogger(OrmLearnApplication.class);

@Autowired

private BookService bookService;

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

testQueryMethods();

}

private void testQueryMethods() {

LOGGER.info("Testing Query Methods");

// Example: Search for books containing "Java"

List<Book> booksByTitle = bookService.searchBooksByTitle("Java");

LOGGER.info("Books containing 'Java': {}", booksByTitle);

// Example: Get books published between two dates

List<Book> booksBetweenDates = bookService.getBooksPublishedBetween(new Date(2020, 1, 1), new Date(2022, 1, 1));

LOGGER.info("Books published between 2020 and 2022: {}", booksBetweenDates);

// Example: Get books starting with "Spring"

List<Book> booksStartingWith = bookService.getBooksStartingWith("Spring");

LOGGER.info("Books starting with 'Spring': {}", booksStartingWith);

// Example: Get top 5 recent books

List<Book> topBooks = bookService.getTop5RecentBooks();

LOGGER.info("Top 5 recent books: {}", topBooks);

}

}

1.O/R Mapping:

- We created entities (`Book`, `Author`, `Publisher`) with various relationships using annotations like `@ManyToOne`, `@OneToMany`, and `@JoinColumn`.

- We used `FetchType.LAZY` for `@OneToMany` relationships to optimize performance by loading related entities only when needed.

2. Query Methods:

- We implemented query methods in the `BookRepository` to search for books based on various criteria, such as containing text, starting text, and date ranges.

- We tested these query methods in the `OrmLearnApplication` class, logging the results to verify their functionality.

This setup provides a solid foundation for a library management application using Spring Data JPA, showcasing both O/R mapping and query capabilities.

* **Objectives**  
  Demonstrate writing Hibernate Query Language and Native Query
  + HQL stands for Hibernate Query Language, JPQL stands for Java Persistence Query Language, Compare HQL and JPQL, @Query annotation, HQL fetch keyword, aggregate functions in HQL, Native Query, nativeQuery attribute
    - Reference - https://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/ch11.html
    - Features of JPA Query - https://www.baeldung.com/spring-data-jpa-query

To demonstrate writing Hibernate Query Language (HQL) and Native Queries in a Spring Data JPA application, we will extend the previous library management application. This will include examples of HQL, JPQL, and Native Queries, as well as the use of the `@Query` annotation, aggregate functions, and the `fetch` keyword.

Step 1: Understanding HQL and JPQL

HQL (Hibernate Query Language): HQL is an object-oriented query language that is similar to SQL but operates on the entity objects rather than the database tables. It allows you to perform queries on the entity model.

- JPQL (Java Persistence Query Language): JPQL is a standard query language defined by the JPA specification. It is similar to HQL but is designed to be used with any JPA provider, not just Hibernate.

- Comparison:

- HQL is specific to Hibernate, while JPQL is part of the JPA specification.

- Both HQL and JPQL operate on entity objects rather than database tables.

- HQL supports Hibernate-specific features, while JPQL adheres to the JPA standard.

Step 2: Implementing HQL and JPQL in the Repository

1. Update the `BookRepository` Interface:

```java

package com.cognizant.springlearn.repository;

import com.cognizant.springlearn.model.Book;

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

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

import org.springframework.data.repository.query.Param;

import java.util.Date;

import java.util.List;

public interface BookRepository extends JpaRepository<Book, Long> {

// HQL Example: Find books by title using HQL

@Query("SELECT b FROM Book b WHERE b.title LIKE %:title%")

List<Book> findByTitleContainingHQL(@Param("title") String title);

// JPQL Example: Find books published between two dates

@Query("SELECT b FROM Book b WHERE b.publishedDate BETWEEN :startDate AND :endDate")

List<Book> findByPublishedDateBetweenJPQL(@Param("startDate") Date startDate, @Param("endDate") Date endDate);

// HQL Example: Count total books

@Query("SELECT COUNT(b) FROM Book b")

Long countTotalBooksHQL();

// Native Query Example: Find books by title using native SQL

@Query(value = "SELECT \* FROM book WHERE title LIKE %:title%", nativeQuery = true)

List<Book> findByTitleContainingNative(@Param("title") String title);

}

```

Step 3: Using Fetch Keyword and Aggregate Functions

1. Update the `BookService` Class:

```java

package com.cognizant.springlearn.service;

import com.cognizant.springlearn.model.Book;

import com.cognizant.springlearn.repository.BookRepository;

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

import org.springframework.stereotype.Service;

import java.util.Date;

import java.util.List;

@Service

public class BookService {

@Autowired

private BookRepository bookRepository;

public List<Book> searchBooksByTitleHQL(String title) {

return bookRepository.findByTitleContainingHQL(title);

}

public List<Book> getBooksPublishedBetweenJPQL(Date startDate, Date endDate) {

return bookRepository.findByPublishedDateBetweenJPQL(startDate, endDate);

}

public Long countTotalBooksHQL() {

return bookRepository.countTotalBooksHQL();

}

public List<Book> searchBooksByTitleNative(String title) {

return bookRepository.findByTitleContainingNative(title);

}

}

Step 4: Testing HQL and Native Queries

1. Update the `OrmLearnApplication` Class:

```java

package com.cognizant.springlearn;

import com.cognizant.springlearn.model.Book;

import com.cognizant.springlearn.service.BookService;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.Date;

import java.util.List;

@SpringBootApplication

public class OrmLearnApplication implements CommandLineRunner {

private static final Logger LOGGER = LoggerFactory.getLogger(OrmLearnApplication.class);

@Autowired

private BookService bookService;

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

testHQLAndNativeQueries();

}

private void testHQLAndNativeQueries() {

LOGGER.info("Testing HQL and Native Queries");

// Example: Search for books containing "Java" using HQL

List<Book> booksByTitleHQL = bookService.searchBooksByTitleHQL("Java");

LOGGER.info("Books containing 'Java' (HQL): {}", booksByTitleHQL);

// Example: Get books published between two dates using JPQL

List<Book> booksBetweenDatesJPQL = bookService.getBooksPublishedBetweenJPQL(new Date(2020, 1, 1), new Date(2022, 1, 1));

LOGGER.info("Books published between 2020 and 2022 (JPQL): {}", booksBetweenDatesJPQL);

// Example: Count total books using HQL

Long totalBooksHQL = bookService.countTotalBooksHQL();

LOGGER.info("Total books (HQL): {}", totalBooksHQL);

// Example: Search for books containing "Spring" using Native Query

List<Book> booksByTitleNative = bookService.searchBooksByTitleNative("Spring");

LOGGER.info("Books containing 'Spring' (Native Query): {}", booksByTitleNative);

}

}

```

1. HQL and JPQL:

- We created query methods in the `BookRepository` using HQL and JPQL to search for books by title and published dates.

- We used the `@Query` annotation to define custom queries.

2. Aggregate Functions:

- We implemented an aggregate function to count the total number of books using HQL.

3. Native Queries:

- We demonstrated the use of native SQL queries to search for books by title.

4. Testing:

- We tested the HQL and Native Queries in the `OrmLearnApplication`, logging the results to verify their functionality.