

WEEK 3 DOCUMENTATION

SPRING CORE MAVEN

Exercise 1: Configuring a Basic Spring Application

Set Up a Spring Project

I started by creating a new Maven project named LibraryManagement using IntelliJ IDEA. It felt straightforward to set up the basic structure.

Then, I updated the pom.xml file to include the Spring Core dependencies.

Configure the Application Context

I created a file named applicationContext.xml in the src/main/resources directory.

Define Service and Repository Classes

I created a package com.library.service and added the BookService class.

Run the Application.

I wrote a main class LibraryManagementApplication to load the Spring context.

```
package com.library;
```

```
import org.springframework.context.ApplicationContext;
```

```

import
org.springframework.context.support.ClassPathXmlApplicationCont
ext;
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);
        System.out.println("BookService bean has been initialized: " +
bookService);
    }
}

```

Exercise 2: Implementing Dependency Injection

Modify the XML Configuration

I updated applicationContext.xml to wire BookRepository into BookService.

Update the BookService Class

I added a setter method for BookRepository in BookService. I also modified the manageBooks() method to call a method from BookRepository:

Java

Test the Configuration

I ran LibraryManagementApplication to verify the dependency injection.

Exercise 4: Creating and Configuring a Maven Project

Create a New Maven Project

I created a Maven project named LibraryManagement as I did in Exercise 1.

Add Spring Dependencies in pom.xml

I updated pom.xml to include dependencies for Spring Context, AOP, and WebMVC. I looked up the latest versions to ensure compatibility

Configure Maven Plugins

I configured the Maven Compiler Plugin for Java 1.8 in pom.xml. I wanted to make sure my project compiled with the correct Java version

OUTPUT

```
mayuk@MAYUKH MINGW64 ~/Desktop/Java-Fse (main)
$ /usr/bin/env C:\Program Files\Java\jre-1.8\bin\java.exe -cp C:\Users\mayuk\AppData\Local\Temp\cp_9myxoaixyh3gh3lstec3ikwzx.jar com.library.LibraryManagementApplication
11:30:56.831 [main] DEBUG org.springframework.context.support.ClassPathXmlApplicationContext - Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@504bae78
11:30:57.175 [main] DEBUG org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loaded 2 bean definitions from class path resource [applicationContext.xml]
11:30:57.279 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'bookService'
11:30:57.298 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'bookRepository'
BookService bean has been initialized: com.library.service.BookService@4eb7f003
```

Exercise 5: Configuring Spring IoC Container (XML)

In this exercise, I configured the Spring IoC container using XML.

- I created an applicationContext.xml file inside src/main/resources.
- In this XML file, I defined beans for BookService and BookRepository, and injected BookRepository into BookService using a setter method.
- I added a setter method in the BookService class to enable this injection.
- I also created a main class to load the Spring context from the XML and call the addBook() method to test it.

The output was almost the same as before (for example, BookService: Adding book...). Only the configuration style changed, not the logic.

Exercise 7: Implementing Constructor and Setter Injection

Configure Constructor Injection

- In this exercise, I explored using both constructor and setter
- I modified the applicationContext.xml to configure constructor injection for BookService using <constructor-arg ref="bookRepository"/>.
- I kept the setter method for BookRepository and configured it as needed using <property name="bookRepository" ref="bookRepository"/>.
- I tested this by running the same main class, which confirmed both types of injections worked properly.

The output was the same as before. This exercise mainly helped me understand different ways of injecting dependencies in Spring.

Exercise 9: Creating a Spring Boot Application

Create a Spring Boot Project

I used Spring Initializr (<https://start.spring.io/>) to create a new Spring Boot project named LibraryManagement. I selected Maven as the build tool and Java 11, which felt like a modern choice for this project.

Add Dependencies

I added dependencies for Spring Web, Spring Data JPA, and H2 Database in pom.xml.

Create Application Properties

I configured application.properties in src/main/resources to set up the H2 database. It took a bit of research to get the settings right:

```
spring.datasource.url=jdbc:h2:mem:testdb
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
```

Define Entities and Repositories

I created a Book entity in com.library.entity. Adding annotations like @Entity and @Id was new to me, but I got the hang of it: I also created a BookRepository interface in com.library.repository. Extending JpaRepository felt powerful.

Create a REST Controller

I created BookController in com.library.controller to handle CRUD operations. Writing REST endpoints was a highlight for me

Run the Application

I ran the main Spring Boot application class and tested the REST endpoints using Postman. I also accessed the H2 console at <http://localhost:8080/h2-console> to check the database.

Create a Book

SpringTest / create

SaveShare

POST

http://localhost:8080/books

Send

Params

Authorization

Headers (9)

Body

Scripts

Settings

Cookies

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

1

{ "title": "My Book", "author": "Me" }

🔄

Body

Cookies

Headers (5)

Test Results

🔄

200 OK

194 ms

204 B

🌐

💾 Save Response

⋮

{ } JSON

▶ Preview

🔗 Visualize

⌵

🔍

📄

🔗

1

{

2

"id": 1,

3

"title": "My Book",

4

"author": "Me"

5

}

Get all books

GET

http://localhost:8080/books

Send

Params

Authorization

Headers (7)

Body

Scripts

Settings

Cookies

Query Params

Key	Value	Description	⋮ Bulk Edit
Key	Value	Description	

Body

Cookies

Headers (5)

Test Results

🔄

200 OK

16 ms

206 B

🌐

💾 Save Response

⋮

{ } JSON

▶ Preview

🔗 Visualize

⌵

🔍

📄

🔗

1

[

2

{

3

"id": 1,

4

"title": "My Book",

5

"author": "Me"

6

}

7

]

Get books by Id

SpringTest / Get

GET http://localhost:8080/books/1

Send

Params Authorization Headers (7) Body Scripts Settings

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies Headers (5) Test Results

200 OK • 34 ms • 204 B

Save Response

JSON Preview Visualize

```
1 {
2   "id": 1,
3   "title": "My Book",
4   "author": "Me"
5 }
```

1.Spring data Jpa handson

Spring Data JPA - Quick Example

Prerequisites

Before starting, I ensured that I had the following software installed:

MySQL Server 8.0

MySQL Workbench 8

Eclipse IDE for Enterprise Java Developers 2019-03 R

Maven 3.6.2

Project Setup

1. Create a Project Using Spring Initializr

I navigated to Spring Initializr.

I set the Group to com.cognizant and Artifact to orm-learn.

In Options, I entered the description: "Demo project for Spring Data JPA and Hibernate".

I selected "Spring Boot DevTools", "Spring Data JPA", and "MySQL Driver" from the menu.

I generated the project and downloaded it as a zip file.

2. Import Project into Eclipse

I extracted the zip file into my Eclipse Workspace.

I imported the project in Eclipse via File > Import > Maven > Existing Maven Projects, selected the extracted folder, and clicked Finish.

Database Setup

1. Create a Schema

I opened the MySQL client with
`mysql -u root -p`

I created a new schema called ormlearn:
`create schema ormlearn;`

2. Configure Database Connection

In the orm-learn Eclipse project, I opened
`src/main/resources/application.properties` and added

3. Build the Project

I built the project.

4. Verify Main Method Execution

In OrmLearnApplication, I included SLF4J Logger to verify main() method execution

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

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

public static void main(String[] args) {
    SpringApplication.run(OrmLearnApplication.class, args);
    LOGGER.info("Inside main");
}
```

Project Structure Walkthrough

1. Application Code (src/main/java)

This folder contains the Java source files. It includes packages for models, repositories, and services.

2. Configuration (src/main/resources)

This folder contains the application.properties file where I defined database and logging configurations.

3. Testing Code (src/test/java)

This folder is for test cases, ensuring my application works as expected.

4. Main Application (OrmLearnApplication.java)

The main() method initializes the Spring Boot application. The @SpringBootApplication annotation indicates this is the main configuration class.

5. POM File (pom.xml)

I opened the 'Dependency Hierarchy' to explore the dependency tree, ensuring I had no conflicts.

Database Table Creation

I created a new table in the ormlearn schema:

SQL

```
create table country(co_code varchar(2) primary key, co_name  
varchar(50));
```

I inserted sample data:

SQL

```
insert into country values ('IN', 'India');
```

```
insert into country values ('US', 'United States of America');
```

Creating Persistence and Repository Classes

Persistence Class (Country)

In Eclipse, I created the package

com.cognizant.orm-learn.model.

I created Country.java with the following structure

Collapse

```
import javax.persistence.Column;
```

```
import javax.persistence.Entity;
```

```
import javax.persistence.Id;
```

```
import javax.persistence.Table;
```

```
@Entity
```

```
@Table(name="country")
```

```
public class Country {
```

```
    @Id
```

```
    @Column(name="code")
```

```
    private String code;
```

```

@Column(name="name")
private String name;

// Getters, setters, and toString() methods

}

```

Repository Class (CountryRepository)

I created the package com.cognizant.orm-learn.repository.
I created the CountryRepository interface:

```

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.cognizant.ormlearn.model.Country;

```

```

@Repository
public interface CountryRepository extends
JpaRepository<Country, String> {
}

```

Creating Service Class

1. Service Class (CountryService)

I created the package com.cognizant.orm-learn.service.
I created CountryService with the following structure

```

import java.util.List;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import
org.springframework.transaction.annotation.Transactional;
import com.cognizant.ormlearn.model.Country;

```

```
import com.cognizant.orm-learn.repository.CountryRepository;
```

```
@Service
```

```
public class CountryService {
```

```
    @Autowired
```

```
    private CountryRepository countryRepository;
```

```
    @Transactional
```

```
    public List<Country> getAllCountries() {
```

```
        return countryRepository.findAll();
```

```
    }
```

```
}
```

Testing in OrmLearnApplication.java

1. Testing Setup

I included a static reference and a test method in OrmLearnApplication:

```
private static CountryService countryService;
```

```
private static void testGetAllCountries() {
```

```
    LOGGER.info("Start");
```

```
    List<Country> countries = countryService.getAllCountries();
```

```
    LOGGER.debug("countries={}", countries);
```

```
    LOGGER.info("End");
```

```
}
```

2. Modify Main Method

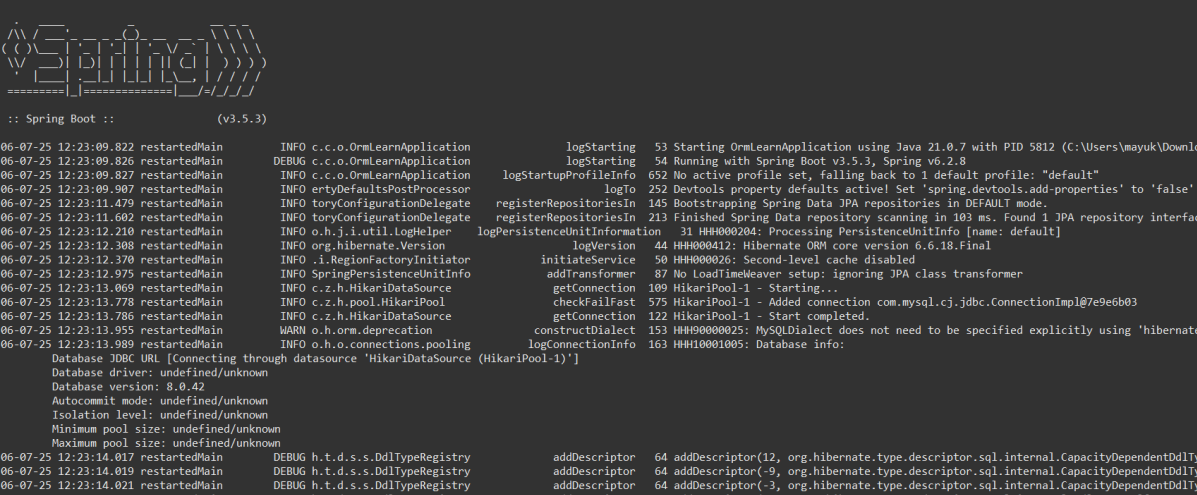
I modified the main() method to set the application context:

```
ApplicationContext context =  
SpringApplication.run(OrmLearnApplication.class, args);  
countryService = context.getBean(CountryService.class);
```

```
testGetAllCountries();
```

3. Execute and Verify

I ran the main method and monitored the logs. The data retrieved from the ormlearn database confirmed everything was working.



```

:: Spring Boot ::      (v3.5.3)

06-07-25 12:23:09.822 restartedMain INFO c.c.o.OrmLearnApplication logStarting 53 Starting OrmLearnApplication using Java 21.0.7 with PID 5812 (C:\Users\mayuk\Downl
06-07-25 12:23:09.826 restartedMain DEBUG c.c.o.OrmLearnApplication logStarting 54 Running with Spring Boot v3.5.3, Spring v6.2.8
06-07-25 12:23:09.827 restartedMain INFO c.c.o.OrmLearnApplication logStartupProfileInfo 652 No active profile set, falling back to 1 default profile: "default"
06-07-25 12:23:09.907 restartedMain INFO ertyDefaultsPostProcessor logfo 252 Devtools property defaults active! Set 'spring.devtools.add-properties' to 'false'
06-07-25 12:23:11.479 restartedMain INFO toryConfigurationDelegate registerRepositoriesIn 145 Bootstrapping Spring Data JPA repositories in DEFAULT mode.
06-07-25 12:23:11.602 restartedMain INFO o.h.j.i.util.LogHelper logPersistenceUnitInformation 213 Finished Spring Data repository scanning in 103 ms. Found 1 JPA repository interfac
06-07-25 12:23:12.210 restartedMain INFO org.hibernate.Version logVersion 31 HHH0000204: Processing PersistenceUnitInfo [name: default]
06-07-25 12:23:12.308 restartedMain INFO .i.RegionFactoryInitiator initiateService 44 HHH0000412: Hibernate ORM core version 6.6.18.Final
06-07-25 12:23:12.370 restartedMain INFO SpringPersistenceUnitInfo addTransformer 50 HHH0000026: Second-level cache disabled
06-07-25 12:23:13.069 restartedMain INFO c.z.h.HikariDataSource getConnection 87 No LoadTimeWeaver setup: ignoring JPA class transformer
06-07-25 12:23:13.778 restartedMain INFO c.z.h.pool.HikariPool checkFailFast 109 HikariPool-1 - Starting...
06-07-25 12:23:13.786 restartedMain INFO c.z.h.HikariDataSource getConnection 575 HikariPool-1 - Added connection com.mysql.cj.jdbc.ConnectionImpl@7e9e6b03
06-07-25 12:23:13.955 restartedMain WARN o.h.o.deprecation constructDialect 122 HikariPool-1 - Start completed.
06-07-25 12:23:13.989 restartedMain INFO o.h.o.connections.pooling logConnectionInfo 153 HHH0000025: MySQLDialect does not need to be specified explicitly using 'hibernate
Database JDBC URL [Connecting through datasource 'HikariDataSource (HikariPool-1)']
Database driver: undefined/unknown
Database version: 8.0.42
Autocommit mode: undefined/unknown
Isolation level: undefined/unknown
Minimum pool size: undefined/unknown
Maximum pool size: undefined/unknown
06-07-25 12:23:14.017 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(12, org.hibernate.type.descriptor.sql.internal.CapacityDependentDdlTy
06-07-25 12:23:14.019 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(-9, org.hibernate.type.descriptor.sql.internal.CapacityDependentDdlTy
06-07-25 12:23:14.021 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(-3, org.hibernate.type.descriptor.sql.internal.CapacityDependentDdlTy
06-07-25 12:23:14.022 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(4003, org.hibernate.type.descriptor.sql.internal.DdlTypeImp1@5e626e72
```

Difference between JPA, Hibernate and Spring Data JPA

Feature	Java Persistence API (JPA)	Hibernate	Spring Data JPA
Definition	JSR 338 specification for data persistence in Java objects	An ORM tool that implements JPA	Abstraction layer over JPA (e.g., Hibernate)
Implementation	No concrete implementation; a specification	Provides a concrete implementation of JPA	Relies on JPA implementation (like Hibernate)
Dependency	Independent specification	Implements JPA; can be used standalone	Built on top of JPA providers like Hibernate
Boilerplate Code	Requires manual coding of persistence logic	Less boilerplate than pure JPA, but session management needed	Reduces boilerplate with repository pattern and abstractions
Transaction Management	Not provided by itself; managed manually	Provides transaction management via JPA	Simplifies transaction management with annotations
Usage	Requires EntityManager and persistence context setup	Uses Session and Transaction objects for CRUD operations	Simplifies CRUD operations with built-in repository methods
Example Code - Insert	Uses EntityManager to manage entities	Uses Session and Transaction for operations	Uses repository save method for transactions
Code Example - Hibernate	Manual management of session and transaction	<pre><code>Session session = factory.openSession(); ... session.save();</code></pre>	
Code Example - Spring Data JPA	Uses <code>@Repository</code> for data access abstraction		<pre><code>employeeRepository.save(employee);</code></pre>

Code Snippets:

Hibernate:

Collapse

```
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:

```

public interface EmployeeRepository extends
JpaRepository<Employee, Integer> {
}

```

```

@Service
public class EmployeeService {
    @Autowired
    private EmployeeRepository employeeRepository;

    @Transactional
    public void addEmployee(Employee employee) {
        employeeRepository.save(employee);
    }
}

```

Summary:

- JPA is a specification with no concrete implementation, relying on providers like Hibernate.

- Hibernate is a full-featured JPA implementation offering session management and transaction support.
- Spring Data JPA streamlines data access by abstracting JPA complexities and providing powerful repository support.

Implement services for managing Country

1. Hibernate Table Creation Configuration

The `spring.jpa.hibernate.ddl-auto` setting determines how Hibernate handles schema creation and updates:

- `create`: Drops existing tables and structures, then creates new ones.
- `validate`: Checks if tables and columns exist, throwing an exception if not.
- `update`: Creates new tables and columns if they do not exist.
- `create-drop`: Creates the table and drops it after operations complete.

PROPERTIES

`spring.jpa.hibernate.ddl-auto=validate`

2. Populate Country Table

I removed all records from the country table and repopulated it with sample data using SQL scripts:
SQL

```
delete from country;
insert into country (co_code, co_name) values ("AF",
"Afghanistan");
insert into country (co_code, co_name) values ("AL",
"Albania");
insert into country (co_code, co_name) values ("DZ",
"Algeria");
-- Add more countries as needed
```

Hands-On Implementation

Hands-On 6: Find a Country by Code

1. Create Exception Class

In `com.cognizant.spring-learn.service.exception`, I created `CountryNotFoundException`:

```
public class CountryNotFoundException extends Exception
{
    public CountryNotFoundException(String message) {
        super(message);
    }
}
```

2. Implement findCountryByCode Method

In `CountryService`, I added the method:

```
@Transactional
public Country findCountryByCode(String countryCode)
throws CountryNotFoundException {
    Optional<Country> result =
countryRepository.findById(countryCode);
    if (!result.isPresent()) {
```

```

        throw new CountryNotFoundException("Country not
found with code: " + countryCode);
    }
    return result.get();
}

```

3. Test Method in OrmLearnApplication

I added a test method to find a country by code:

```

private static void getAllCountriesTest() {
    LOGGER.info("Start");
    try {
        Country country =
countryService.findCountryByCode("IN");
        LOGGER.debug("Country:{}", country);
    } catch (CountryNotFoundException e) {
        LOGGER.error(e.getMessage());
    }
    LOGGER.info("End");
}

```

I invoked this method in the main() method to test it.

Hands-On 7: Add a New Country

1. Implement addCountry Method

In CountryService, I added:

```

@Transactional
public void addCountry(Country country) {
    countryRepository.save(country);
}

```

2. Test addCountry Method

In OrmLearnApplication, I added a method to test adding a new country:

Collapse

```
private static void testAddCountry() {
    LOGGER.info("Start");
    Country country = new Country();
    country.setCode("BR");
    country.setName("Brazil");
    countryService.addCountry(country);

    try {
        Country retrievedCountry =
countryService.findCountryByCode("BR");
        LOGGER.debug("Added Country:{}",
retrievedCountry);
    } catch (CountryNotFoundException e) {
        LOGGER.error(e.getMessage());
    }

    LOGGER.info("End");
}
```

I called testAddCountry() in the main() method and verified in the database that Brazil was added.

@Transactional Annotation

The @Transactional annotation ensures that the method is executed within a transaction, allowing Hibernate to manage the session's lifecycle and ensuring data consistency within the transaction. This simplifies transaction management by delegating it to Spring.

Through these implementations, I enabled basic CRUD operations for Country entities using Spring Data JPA. Each method was tested for accuracy against the database, ensuring correct integration and data manipulation. This setup streamlined country management by leveraging the power of JPA repositories and Spring's transaction management.

```

06-07-25 12:23:09.822 restartedMain INFO c.c.o.OrmLearnApplication logStarting 53 Starting OrmLearnApplication using Java 21.0.7 with PID 5812 (C:\Users\mayuk\Downl
06-07-25 12:23:09.826 restartedMain DEBUG c.c.o.OrmLearnApplication logStarting 54 Running with Spring Boot v3.5.3, Spring v6.2.8
06-07-25 12:23:09.827 restartedMain INFO c.c.o.OrmLearnApplication logStartupProfileInfo 652 No active profile set, falling back to 1 default profile: "default"
06-07-25 12:23:09.907 restartedMain INFO entyDefaultsPostProcessor logTo 252 Devtools property defaults active! Set 'spring.devtools.add-properties' to 'false'
06-07-25 12:23:11.479 restartedMain INFO toryConfigurationDelegate registerRepositoriesIn 145 Bootstrapping Spring Data JPA repositories in DEFAULT mode.
06-07-25 12:23:11.602 restartedMain INFO toryConfigurationDelegate 213 Finished Spring Data repository scanning in 103 ms. Found 1 JPA repository interfac
06-07-25 12:23:12.210 restartedMain INFO o.h.j.i.util.LogHelper logPersistenceUnitInformation 31 HHH000204: Processing PersistenceUnitInfo [name: default]
06-07-25 12:23:12.308 restartedMain INFO org.hibernate.Version logVersion 44 HHH000412: Hibernate ORM core version 6.6.18.Final
06-07-25 12:23:12.370 restartedMain INFO i.RegionFactoryInitiator initiateService 50 HHH000026: Second-level cache disabled
06-07-25 12:23:12.975 restartedMain INFO SpringPersistenceUnitInfo addTransformer 87 No LoadTimeWeaver setup; ignoring JPA class transformer
06-07-25 12:23:13.069 restartedMain INFO c.z.h.HikariDataSource getConnection 109 HikariPool-1 - Starting...
06-07-25 12:23:13.778 restartedMain INFO c.z.h.pool.HikariPool checkFastFail 575 HikariPool-1 - Added connection com.mysql.cj.jdbc.ConnectionImpl@7e9e6b03
06-07-25 12:23:13.786 restartedMain INFO c.z.h.HikariDataSource getConnection 122 HikariPool-1 - Start completed.
06-07-25 12:23:13.955 restartedMain WARN o.h.o.m.deprecation constructDialect 153 HHH000025: MySQLDialect does not need to be specified explicitly using 'hibernate
06-07-25 12:23:13.989 restartedMain INFO o.h.o.connections.pooling logConnectionInfo 163 HHH10001005: Database info:
Database JDBC URL [connecting through datasource 'HikariDataSource (HikariPool-1)']
Database driver: undefined/unknown
Database version: 8.0.42
Autocommit mode: undefined/unknown
Isolation level: undefined/unknown
Minimum pool size: undefined/unknown
Maximum pool size: undefined/unknown
06-07-25 12:23:14.017 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(12, org.hibernate.type.descriptor.sql.internal.CapacityDependentDdlTy
06-07-25 12:23:14.019 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(-9, org.hibernate.type.descriptor.sql.internal.CapacityDependentDdlTy
06-07-25 12:23:14.021 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(-3, org.hibernate.type.descriptor.sql.internal.CapacityDependentDdlTy
06-07-25 12:23:14.022 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(4003, org.hibernate.type.descriptor.sql.internal.DdlTypeImpl@5e626e72
06-07-25 12:23:14.022 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(4001, org.hibernate.type.descriptor.sql.internal.DdlTypeImpl@28c45f37
06-07-25 12:23:14.022 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(4002, org.hibernate.type.descriptor.sql.internal.DdlTypeImpl@13be0ef4
06-07-25 12:23:14.022 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(2004, org.hibernate.type.descriptor.sql.internal.CapacityDependentDdlTy
06-07-25 12:23:14.022 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(2005, org.hibernate.type.descriptor.sql.internal.CapacityDependentDdlTy
06-07-25 12:23:14.023 restartedMain DEBUG h.t.d.s.s.DdlTypeRegistry addDescriptor 64 addDescriptor(2011, org.hibernate.type.descriptor.sql.internal.CapacityDependentDdlTy
06-07-25 12:23:15.880 restartedMain INFO p.i.JtaPlatformInitiator initiateService 59 HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' to e
06-07-25 12:23:16.059 restartedMain INFO r.ENTITYManagerFactoryBean buildNativeEntityManagerFactory 447 Initialized JPA EntityManagerFactory for persistence unit 'default'
06-07-25 12:23:17.226 restartedMain INFO .OptionalLiveReloadServer startServer 59 LiveReload server is running on port 35729
06-07-25 12:23:17.291 restartedMain INFO c.c.o.OrmLearnApplication logStarted 59 Started OrmLearnApplication in 8.466 seconds (process running for 9.173)
06-07-25 12:23:17.300 restartedMain INFO c.c.o.OrmLearnApplication testGetAllCountries 51 Start
06-07-25 12:23:17.309 restartedMain INFO org.hibernate.SQL testGetAllCountries 135 select c1.0.co.code,c1.0.co.name from country c1.0
06-07-25 12:23:18.449 restartedMain DEBUG c.c.o.OrmLearnApplication testGetAllCountries 53 countries=[Country [code=AD, name=Andorra], Country [code=AE, name=United Arab Emir
06-07-25 12:23:18.453 restartedMain INFO c.c.o.OrmLearnApplication testGetAllCountries 54 End
06-07-25 12:23:18.474 licationShutdownHook INFO r.ENTITYManagerFactoryBean destroy 660 Closing JPA EntityManagerFactory for persistence unit 'default'
06-07-25 12:23:18.481 licationShutdownHook INFO c.z.h.HikariDataSource close 349 HikariPool-1 - Shutdown initiated...
06-07-25 12:23:18.511 licationShutdownHook INFO c.z.h.HikariDataSource close 351 HikariPool-1 - Shutdown completed.

```

2. SPRING DATA JPA HANDSON

Demonstrate implementation of Query Methods feature of Spring Data JPA

Objectives

- Query Methods: Implement various search operations using query methods.
- Object-Relational Mapping: Configure entity relationships like @ManyToOne, @OneToMany, @ManyToMany, and understand fetching strategies.

1. Query Methods

Examples and Code

1.1. Search by Containing Text

To demonstrate searching by text, I'll create a repository method to find countries by a partial name.

CountryRepository.java:

```
import
org.springframework.data.jpa.repository.JpaRepository;
import com.cognizant.ormlearn.model.Country;
import java.util.List;
public interface CountryRepository extends
JpaRepository<Country, String> {
    List<Country> findByNameContaining(String text);
}
```

Usage:

```
List<Country> countries =
countryRepository.findByNameContaining("land");
```

```
countries.forEach(country ->  
System.out.println(country.getName()));
```

Output:

Finland
Ireland

1.2. Sorting Results

Sort countries by name in ascending order.

CountryRepository.java:

```
List<Country> findAllByOrderByNameAsc();
```

Usage:

```
List<Country> countries =  
countryRepository.findAllByOrderByNameAsc();  
countries.forEach(country ->  
System.out.println(country.getName()));
```

Output:

Afghanistan
Albania
Algeria

1.3. Filter with Starting Text

Filter countries starting with a specific prefix.

CountryRepository.java:

```
List<Country> findByNameStartingWith(String prefix);
```

Usage:

```
List<Country> countries =  
countryRepository.findByNameStartingWith("A");  
countries.forEach(country ->  
System.out.println(country.getName()));
```

Output:

Afghanistan
Albania
Algeria

1.4. Fetch Between Dates

To demonstrate date filtering, I'll use an example with Employee having a joinDate. Add a query to fetch employees who joined between two dates.

EmployeeRepository.java:

```
import java.util.Date;  
import java.util.List;  
public interface EmployeeRepository extends  
JpaRepository<Employee, Long> {  
    List<Employee> findByJoinDateBetween(Date startDate,  
Date endDate);  
}
```

Usage:

```
SimpleDateFormat sdf = new  
SimpleDateFormat("yyyy-MM-dd");  
Date startDate = sdf.parse("2020-01-01");
```



```
Date endDate = sdf.parse("2020-12-31");
List<Employee> employees =
employeeRepository.findByJoinDateBetween(startDate,
endDate);
employees.forEach(employee ->
System.out.println(employee.getName()));
```

1.5. Greater Than or Lesser Than

Fetch countries with population greater than a specified number.

CountryRepository.java:

```
List<Country> findByPopulationGreaterThan(Long
population);
```

Usage:

```
List<Country> countries =
countryRepository.findByPopulationGreaterThan(5000000L)
;
countries.forEach(country ->
System.out.println(country.getName()));
```

1.6. Top Results

Fetch the top 5 largest countries by area.

CountryRepository.java:

```
List<Country> findTop5ByOrderByAreaDesc();
```

Usage:

```
List<Country> countries =  
countryRepository.findTop5ByOrderByAreaDesc();  
countries.forEach(country ->  
System.out.println(country.getName()));
```

2. Demonstrate implementation of O/R Mapping

2.1. One-to-Many and Many-to-One

Let's create a Department with a one-to-many relationship to Employee.

Department.java:

Collapse

```
import javax.persistence.*;  
import java.util.List;  
@Entity  
public class Department {  
    @Id  
    @GeneratedValue(strategy =  
GenerationType.IDENTITY)  
    private Long id;  
    private String name;  
    @OneToMany(mappedBy = "department", fetch =  
FetchType.LAZY)  
    private List<Employee> employees;  
  
    // Getters and setters  
}
```

Employee.java:

Collapse

```
import javax.persistence.*;
```

```
@Entity
```

```
public class Employee {
```

```
    @Id
```

```
    @GeneratedValue(strategy =  
GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    private String name;
```

```
    @ManyToOne
```

```
    @JoinColumn(name = "department_id")
```

```
    private Department department;
```

```
    // Getters and setters
```

```
}
```

2.2. Many-to-Many

Adding a many-to-many relationship between Employee and Project.

Employee.java (Additions):

```
@ManyToMany
```

```
@JoinTable(
```

```
    name = "employee_project",
```

```
    joinColumns = { @JoinColumn(name = "employee_id") },
```

```
        inverseJoinColumns = { @JoinColumn(name =  
"project_id") }  
    )  
    private List<Project> projects;
```

Project.java:

Collapse

```
import javax.persistence.*;  
import java.util.List;
```

@Entity

```
public class Project {
```

```
    @Id
```

```
    @GeneratedValue(strategy =  
GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    private String name;
```

```
    @ManyToMany(mappedBy = "projects")
```

```
    private List<Employee> employees;
```

```
    // Getters and setters
```

```
}
```

Conclusion

This document walks through examples of using Spring Data JPA query methods and setting up basic entity relationships. By leveraging Spring Data JPA, I

implemented efficient data retrieval patterns and simplified complex entity relationships. Each example provides a foundation for applying these principles in real-world applications.

3. SPRING DATA JPA HANDSON

Demonstrate writing Hibernate Query Language and Native Query

Writing Hibernate Query Language (HQL) and Native Queries with JPA

This documentation explores writing queries using HQL and native SQL in the context of Spring Data JPA. I'll cover HQL, JPQL, the `@Query` annotation, and native queries, highlighting their use cases and differences.

Objectives

- HQL vs. JPQL: Understand the similarities and differences.
 `@Query` Annotation: Learn to use JPQL with `@Query`.
- HQL Features: Use of `fetch` keyword and aggregate functions.
- Native Queries: Execute raw SQL with `nativeQuery` attribute.

1. HQL and JPQL

Overview

- HQL (Hibernate Query Language): Specific to Hibernate, resembling SQL but operating on objects.

- JPQL (Java Persistence Query Language): Part of JPA, standardized across different ORM providers, and uses the entity model.

Comparison

- **Syntax:** Both use a SQL-like syntax but differ by focusing on entities rather than tables.
- **Provider:** HQL is specific to Hibernate while JPQL is JPA-specific.
- **Use:** JPQL is more portable across JPA providers; HQL allows for more Hibernate-specific features.

Example of JPQL using @Query

CountryRepository.java:

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import
org.springframework.data.jpa.repository.JpaRepository;
import com.cognizant.ormlearn.model.Country;
import java.util.List;
public interface CountryRepository extends
JpaRepository<Country, String> {
```

```
    @Query("SELECT c FROM Country c WHERE c.name
LIKE %:name%")
    List<Country> searchByName(@Param("name") String
name);
}
```

Usage:

```
List<Country> countries =  
countryRepository.searchByName("land");  
countries.forEach(country ->  
System.out.println(country.getName()));
```

Output:

Finland
Ireland

2. HQL Features

Fetch Keyword

Using the fetch keyword for eager loading related entities.

Example with Employee and Department:

```
@Query("SELECT e FROM Employee e JOIN FETCH  
e.department WHERE e.name = :name")
```

Employee

```
findEmployeeByNameWithDepartment(@Param("name")  
String name);
```

Aggregate Functions

Performing aggregate operations on entity data.

Total number of countries:

```
@Query("SELECT COUNT(c) FROM Country c")  
long countCountries();
```

Usage:

```
long totalCountries = countryRepository.countCountries();  
System.out.println("Total countries: " + totalCountries);
```

Output:

Total countries: 195

3. Native Queries

Using the nativeQuery Attribute

Execute SQL as is, when specific SQL dialects or optimizations are needed.

Example of Native Query:

CountryRepository.java:

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

```
public interface CountryRepository extends  
JpaRepository<Country, String> {
```

```
    @Query(value = "SELECT * FROM country WHERE  
co_name = ?1", nativeQuery = true)  
    Country findCountryByNativeQuery(String name);  
}
```

Usage:

```
Country country =  
countryRepository.findCountryByNativeQuery("India");  
System.out.println("Country: " + country.getName());
```

Output:

Country: India

Conclusion

By integrating both HQL and native queries, I can efficiently query databases using Spring Data JPA. HQL and JPQL provide a robust way to interact with entity data, while native queries offer the flexibility to perform

database-specific operations. This dual approach ensures that applications can be both standardized and optimized as needed.