**WEEK-3(HANDS ON)**

**Spring Core And Maven**

**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.

**Source code:**

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.library</groupId>

<artifactId>LibraryManagement</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<!-- Spring Context dependency -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.31</version>

</dependency>

</dependencies>

</project>

xml

Copy

Edit

<!-- src/main/resources/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

http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- Bean for BookRepository -->

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

<!-- Bean for BookService with DI -->

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

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

</bean>

</beans>

Copy

Edit

// src/main/ /com/library/repository/BookRepository.

package com.library.repository;

public class BookRepository {

public void saveBook(String bookName) {

System.out.println("BookRepository: Saving book - " + bookName);

}

}

Copy

Edit

// src/main/ /com/library/service/BookService.

package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

private BookRepository bookRepository;

// Setter for dependency injection

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public void addBook(String bookName) {

System.out.println("BookService: Adding book - " + bookName);

bookRepository.saveBook(bookName);

}

}

Copy

Edit

// src/main/ /com/library/LibraryApp.

package com.library;

import com.library.service.BookService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LibraryApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

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

bookService.addBook("The Alchemist");

((ClassPathXmlApplicationContext) context).close();

}

}

**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.

**Sorce code:**

// pom.xml (Maven configuration)

<!--

<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>5.3.31</version>

</dependency>

</dependencies>

</project>

-->

// BookRepository.

package com.library.repository;

public class BookRepository {

public void save(String bookName) {

System.out.println("Saving: " + bookName);

}

}

// BookService.

package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

private BookRepository bookRepository;

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public void addBook(String bookName) {

bookRepository.save(bookName);

}

}

// LibraryManagementApplication. (main class)

package com.library;

import com.library.service.BookService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LibraryManagementApplication {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

BookService service = context.getBean("bookService", BookService.class);

service.addBook("Wings of Fire");

((ClassPathXmlApplicationContext) context).close();

}

}

**Exercise 3: Implementing Logging with Spring AOP**

**Scenario:**

The library management application requires logging capabilities to track method execution times.

**Source code:**

// pom.xml (include AOP + Core)

/\*

<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>5.3.31</version>

</dependency>

<!-- Spring AOP -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.31</version>

</dependency>

<!-- AspectJ Weaver (required for proxying) -->

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.9.21</version>

</dependency>

</dependencies>

</project>

\*/

// applicationContext.xml (Enable AOP and register beans)

/\*

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:aop="http://www.springframework.org/schema/aop"

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

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- Enable AOP support -->

<aop:aspectj-autoproxy />

<!-- Declare repository and service -->

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

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

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

</bean>

<!-- Register Logging Aspect -->

<bean id="loggingAspect" class="com.library.aspect.LoggingAspect"/>

</beans>

\*/

// BookRepository.

package com.library.repository;

public class BookRepository {

public void save(String bookName) {

System.out.println("Saving book: " + bookName);

}

}

// BookService.

package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

private BookRepository bookRepository;

public void setBookRepository(BookRepository repo) {

this.bookRepository = repo;

}

public void addBook(String bookName) {

System.out.println("Adding book: " + bookName);

bookRepository.save(bookName);

}

}

// LoggingAspect.

package com.library.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.\*;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class LoggingAspect {

@Around("execution(\* com.library.service.\*.\*(..))")

public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {

long start = System.currentTimeMillis();

Object result = joinPoint.proceed(); // method call

long duration = System.currentTimeMillis() - start;

System.out.println( joinPoint.getSignature() + " executed in " + duration + "ms");

return result;

}

}

// LibraryManagementApplication.

package com.library;

import com.library.service.BookService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LibraryManagementApplication {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

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

bookService.addBook("Wings of Fire");

((ClassPathXmlApplicationContext) context).close();

}

}

**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

**source code:**

<!-- pom.xml -->

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.library</groupId>

<artifactId>LibraryManagement</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<name>Library Management</name>

<!-- version setup -->

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<!-- Spring dependencies -->

<dependencies>

<!-- Spring Core + Context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.31</version>

</dependency>

<!-- Spring AOP -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.31</version>

</dependency>

<!-- AspectJ Weaver (required for AOP) -->

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.9.21</version>

</dependency>

<!-- Spring WebMVC (optional for future web features) -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.31</version>

</dependency>

</dependencies>

<!-- Maven Compiler Plugin -->

<build>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.10.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

**Exercise 5: Configuring the Spring IoC Container**

**Scenario:**

The library management application requires a central configuration for beans and dependencies.

**Source code:**

// applicationContext.xml (placed in src/main/resources)

/\*

<?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>

\*/

// BookRepository.

package com.library.repository;

public class BookRepository {

public void save(String bookName) {

System.out.println(" Book saved: " + bookName);

}

}

// BookService.

package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

private BookRepository bookRepository;

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public void addBook(String name) {

System.out.println(" Adding book: " + name);

bookRepository.save(name);

}

}

// LibraryManagementApplication. (main class)

package com.library;

import com.library.service.BookService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LibraryManagementApplication {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

BookService service = context.getBean("bookService", BookService.class);

service.addBook("Wings of Fire");

((ClassPathXmlApplicationContext) context).close();

}

}

**Exercise 6: Configuring Beans with Annotations**

**Scenario:**

You need to simplify the configuration of beans in the library management application using annotations.

**Source code:**

// applicationContext.xml (placed in src/main/resources)

/\*

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

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

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<!-- Enable annotation scanning for beans -->

<context:component-scan base-package="com.library" />

</beans>

\*/

// BookRepository.

package com.library.repository;

import org.springframework.stereotype.Repository;

@Repository

public class BookRepository {

public void save(String bookName) {

System.out.println(" Book saved: " + bookName);

}

}

// BookService.

package com.library.service;

import com.library.repository.BookRepository;

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

import org.springframework.stereotype.Service;

@Service

public class BookService {

private BookRepository bookRepository;

@Autowired // Setter injection using annotation

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public void addBook(String name) {

System.out.println(" Adding book: " + name);

bookRepository.save(name);

}

}

// LibraryManagementApplication.

package com.library;

import com.library.service.BookService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LibraryManagementApplication {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

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

service.addBook("Wings of Fire");

((ClassPathXmlApplicationContext) context).close();

}

}

**Exercise 7: Implementing Constructor and Setter Injection**

**Scenario:**

The library management application requires both constructor and setter injection for better control over bean initialization.

**Source code:**

// applicationContext.xml (place this in src/main/resources)

/\*

<?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">

<!-- Define BookRepository bean -->

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

<!-- Define BookService bean with constructor and setter injection -->

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

<constructor-arg value="Hyderabad Central Library"/>

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

</bean>

</beans>

\*/

// BookRepository.

package com.library.repository;

public class BookRepository {

public void save(String bookName) {

System.out.println(" Book saved to DB: " + bookName);

}

}

// BookService.

package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

private String libraryName;

private BookRepository bookRepository;

// Constructor injection

public BookService(String libraryName) {

this.libraryName = libraryName;

}

// Setter injection

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public void addBook(String name) {

System.out.println(" [" + libraryName + "] Adding: " + name);

bookRepository.save(name);

}

}

// LibraryManagementApplication. (main class)

package com.library;

import com.library.service.BookService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LibraryManagementApplication {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

BookService service = context.getBean("bookService", BookService.class);

service.addBook("Wings of Fire");

((ClassPathXmlApplicationContext) context).close();

}

}

**Exercise 8: Implementing Basic AOP with Spring**

**Scenario:**

The library management application requires basic AOP functionality to separate cross-cutting concerns like logging and transaction management.

**Source code:**

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

/\*

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:aop="http://www.springframework.org/schema/aop"

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

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- Enable AOP -->

<aop:aspectj-autoproxy />

<!-- Declare beans -->

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

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

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

</bean>

<!-- Register aspect bean -->

<bean id="loggingAspect" class="com.library.aspect.LoggingAspect"/>

</beans>

\*/

// LoggingAspect.

package com.library.aspect;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class LoggingAspect {

@Before("execution(\* com.library.service.BookService.addBook(..))")

public void logBefore() {

System.out.println(" [LOG] Before adding a book...");

}

@After("execution(\* com.library.service.BookService.addBook(..))")

public void logAfter() {

System.out.println(" [LOG] After adding a book.");

}

}

// BookRepository.

package com.library.repository;

public class BookRepository {

public void save(String bookName) {

System.out.println("Book saved: " + bookName);

}

}

// BookService.

package com.library.service;

import com.library.repository.BookRepository;

public class BookService {

private BookRepository bookRepository;

public void setBookRepository(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public void addBook(String name) {

System.out.println(" Adding book: " + name);

bookRepository.save(name);

}

}

// LibraryManagementApplication.

package com.library;

import com.library.service.BookService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LibraryManagementApplication {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

BookService service = context.getBean("bookService", BookService.class);

service.addBook("The Power of Habit");

((ClassPathXmlApplicationContext) context).close();

}

}

**Spring data JPA-Hands on**

**Hands on 1**

**Write queries on country table using Query Methods**Following are the list of queries that is required for an application. Implement these queries using Query Methods feature of Spring Data JPA. Click [here](https://docs.spring.io/spring-data/jpa/docs/2.2.0.RELEASE/reference/html/#jpa.query-methods.query-creation) for reference. Include appropriate methods in OrmLearnApplication and test the same.

**Source code:**

**Step 1:** Update CountryRepository with Query Methods

**Location:** src/main/ /com/cognizant/ormlearn/repository/CountryRepository.

package com.cognizant.ormlearn.repository;

import .util.List;

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> {

// 1. Countries containing a specific substring (case-insensitive)

List<Country> findByNameContainingIgnoreCase(String substring);

// 2. Same as above, but ordered by name ascending

List<Country> findByNameContainingIgnoreCaseOrderByNameAsc(String substring);

// 3. Countries that start with a specific alphabet (case-insensitive)

List<Country> findByNameStartingWithIgnoreCase(String prefix);

}

**Step 2**: Add Methods in CountryService

Location: src/main/ /com/cognizant/ormlearn/service/CountryService.

@Transactional

public List<Country> findByNameContaining(String substring) {

return countryRepository.findByNameContainingIgnoreCase(substring);

}

@Transactional

public List<Country> findByNameContainingSorted(String substring) {

return countryRepository.findByNameContainingIgnoreCaseOrderByNameAsc(substring);

}

@Transactional

public List<Country> findByNameStartingWith(String prefix) {

return countryRepository.findByNameStartingWithIgnoreCase(prefix);

}

**Step 3:** Test Methods in OrmLearnApplication.

Location: src/main/ /com/cognizant/ormlearn/OrmLearnApplication.

Update your main class as follows:

private static void testFindByNameContaining() {

LOGGER.info("Start - testFindByNameContaining");

List<Country> countries = countryService.findByNameContaining("ou");

countries.forEach(country -> LOGGER.debug("Country: {}", country));

LOGGER.info("End - testFindByNameContaining");

}

private static void testFindByNameContainingSorted() {

LOGGER.info("Start - testFindByNameContainingSorted");

List<Country> countries = countryService.findByNameContainingSorted("ou");

countries.forEach(country -> LOGGER.debug("Country: {}", country));

LOGGER.info("End - testFindByNameContainingSorted");

}

private static void testFindByNameStartingWith() {

LOGGER.info("Start - testFindByNameStartingWith");

List<Country> countries = countryService.findByNameStartingWith("Z");

countries.forEach(country -> LOGGER.debug("Country: {}", country));

LOGGER.info("End - testFindByNameStartingWith");

}

Then call these methods from main() after testGetAllCountries():

testFindByNameContaining();

testFindByNameContainingSorted();

testFindByNameStartingWith();

**Hands on 2**

**Write queries on stock table using Query Methods**   
  
With one year stock data of Facebook, Google and Netflix, we need to implement Spring Data JPA Query Methods for the following scenarios:  
  
**Sample Data**  
Sample data for implementing this hands on is provided to you in the platform

**Source code:**

**Step 1: Stock Entity Class**

**Location**: src/main/ /com/cognizant/ormlearn/model/Stock.

package com.cognizant.ormlearn.model;

import jakarta.persistence.\*;

import .util.Date;

@Entity

@Table(name = "stock")

public class Stock {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "st\_id")

private int id;

@Column(name = "st\_code")

private String code;

@Column(name = "st\_date")

@Temporal(TemporalType.DATE)

private Date date;

@Column(name = "st\_open")

private float open;

@Column(name = "st\_close")

private float close;

@Column(name = "st\_volume")

private long volume;

// Getters and Setters

}

**Step 2: StockRepository Interface**

**Location**: src/main/ /com/cognizant/ormlearn/repository/StockRepository.

package com.cognizant.ormlearn.repository;

import com.cognizant.ormlearn.model.Stock;

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

import org.springframework.stereotype.Repository;

import .util.Date;

import .util.List;

@Repository

public interface StockRepository extends JpaRepository<Stock, Integer> {

// 1. Facebook stock in September 2019

List<Stock> findByCodeAndDateBetween(String code, Date startDate, Date endDate);

// 2. Google stocks with price > 1250

List<Stock> findByCodeAndCloseGreaterThan(String code, float price);

// 3. Top 3 highest volume stocks

List<Stock> findTop3ByOrderByVolumeDesc();

// 4. Lowest 3 Netflix stock prices

List<Stock> findTop3ByCodeOrderByCloseAsc(String code);

}

**Step 3: Add Test Methods in OrmLearnApplication.**

**Location**: src/main/ /com/cognizant/ormlearn/OrmLearnApplication.

Inject StockRepository:

@Autowired

private static StockRepository stockRepository;

Add methods:

private static void testFacebookStocksInSept2019() throws ParseException {

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

Date startDate = sdf.parse("2019-09-01");

Date endDate = sdf.parse("2019-09-30");

List<Stock> stocks = stockRepository.findByCodeAndDateBetween("FB", startDate, endDate);

LOGGER.info("Facebook Stocks in Sept 2019:");

stocks.forEach(stock -> LOGGER.debug("{}", stock));

}

private static void testGoogleStocksAbove1250() {

List<Stock> stocks = stockRepository.findByCodeAndCloseGreaterThan("GOOGL", 1250.0f);

LOGGER.info("Google Stocks > 1250:");

stocks.forEach(stock -> LOGGER.debug("{}", stock));

}

private static void testTop3VolumeStocks() {

List<Stock> stocks = stockRepository.findTop3ByOrderByVolumeDesc();

LOGGER.info("Top 3 Stocks by Volume:");

stocks.forEach(stock -> LOGGER.debug("{}", stock));

}

private static void testLowest3NetflixStocks() {

List<Stock> stocks = stockRepository.findTop3ByCodeOrderByCloseAsc("NFLX");

LOGGER.info("Lowest 3 Netflix Stock Prices:");

stocks.forEach(stock -> LOGGER.debug("{}", stock));

}

Then call them from the main method:

testFacebookStocksInSept2019();

testGoogleStocksAbove1250();

testTop3VolumeStocks();

testLowest3NetflixStocks();

**Hands on 4**

**Implement many to one relationship between Employee and Department**   
  
Follow steps below to defined many to one relationship and perform persistence operations:  
**source code:**

**Step 1: Execute payroll.sql to Create Tables**

**In MySQL CLI:**

bash

mysql> source D:\spring-data-jpa-files\payroll.sql

This script will create three tables: employee, department, and skill.

**Step 2: Create Model Classes in com.cognizant.ormlearn.model**

**🔹 Employee.**

package com.cognizant.ormlearn.model;

import jakarta.persistence.\*;

import .util.Date;

@Entity

@Table(name = "employee")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private int id;

@Column(name = "name")

private String name;

@Column(name = "salary")

private double salary;

@Column(name = "permanent")

private boolean permanent;

@Column(name = "date\_of\_birth")

@Temporal(TemporalType.DATE)

private Date dateOfBirth;

// Getters and Setters

@Override

public String toString() {

return "Employee [id=" + id + ", name=" + name +

", salary=" + salary + ", permanent=" + permanent +

", dateOfBirth=" + dateOfBirth + "]";

}

}

**🔹 Department.**

package com.cognizant.ormlearn.model;

import jakarta.persistence.\*;

@Entity

@Table(name = "department")

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private int id;

@Column(name = "name")

private String name;

// Getters and Setters

@Override

public String toString() {

return "Department [id=" + id + ", name=" + name + "]";

}

}

**🔹 Skill.**

package com.cognizant.ormlearn.model;

import jakarta.persistence.\*;

@Entity

@Table(name = "skill")

public class Skill {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private int id;

@Column(name = "name")

private String name;

// Getters and Setters

@Override

public String toString() {

return "Skill [id=" + id + ", name=" + name + "]";

}

}

**Step 3: Create Repository Interfaces in com.cognizant.ormlearn.repository**

**🔹 EmployeeRepository.**

package com.cognizant.ormlearn.repository;

import com.cognizant.ormlearn.model.Employee;

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

import org.springframework.stereotype.Repository;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

**🔹 DepartmentRepository.**

package com.cognizant.ormlearn.repository;

import com.cognizant.ormlearn.model.Department;

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

import org.springframework.stereotype.Repository;

@Repository

public interface DepartmentRepository extends JpaRepository<Department, Integer> {

}

package com.cognizant.ormlearn.repository;

import com.cognizant.ormlearn.model.Skill;

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

import org.springframework.stereotype.Repository;

@Repository

public interface SkillRepository extends JpaRepository<Skill, Integer> {

}

**Hands on 5**

**Implement one to many relationship between Employee and Department**   
  
Department.

* Include new instance variable for set of employees and define the OneToMay annotation

    @OneToMany(mappedBy = "department")

    private Set<Employee> employeeList;

* Include setter and getter for employeeList

OrmLearnApplication.

* Include new method testGetDepartment()
* In this method, get a department using departmentService.get() passing the id. Select an department id that has more than one employee.
* Log the returned department and department.getEmployeeList()
* Include testGetDepartment() method invocation in main method and comment the other test methods.
* Execute the main() method which will fail with LazyInitializationException. This is because the default fetch type for OneToMany relationship is LAZY, hibernate fetches only department details and does not get the employee details.
* In order to get the employee list as well, modify the annotation to include the fetch type as EAGER. Make this change in employeeList annotation definition of Department class.

    @OneToMany(mappedBy = "department", fetch = FetchType.EAGER)

    private Set<Employee> employeeList;

* After this change try executing the main() method, which will fetch both department and employee

**Hands on 6**

**Implement many to many relationship between Employee and Skill**   
**source code:**

**Step 1: Modify Department.**

@OneToMany(mappedBy = "department", fetch = FetchType.EAGER)

private Set<Employee> employeeList;

Add **getter and setter**:

public Set<Employee> getEmployeeList() {

return employeeList;

}

public void setEmployeeList(Set<Employee> employeeList) {

this.employeeList = employeeList;

}

@ManyToOne

@JoinColumn(name = "dp\_id")

private Department department;

**Step 2: Update OrmLearnApplication.**

Import necessary classes:

import com.cognizant.ormlearn.model.Department;

import com.cognizant.ormlearn.service.DepartmentService;

Create testGetDepartment() method:

public static void testGetDepartment() {

Department dept = departmentService.get(1); // Choose department id with multiple employees

LOGGER.debug("Department: {}", dept);

LOGGER.debug("Employees: {}", dept.getEmployeeList());

}

In the main() method:

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

testGetDepartment();

// Comment other test methods

}

After adding fetch = FetchType.EAGER, both department and employees will be loaded successfully.

**Hands-on 6: Many-to-Many – Employee ↔ Skill**

**Step 1: Update Employee.**

Add the skills list:

@ManyToMany(fetch = FetchType.EAGER)

@JoinTable(name = "employee\_skill",

joinColumns = @JoinColumn(name = "es\_em\_id"),

inverseJoinColumns = @JoinColumn(name = "es\_sk\_id"))

private Set<Skill> skillList;

Getter and setter:

public Set<Skill> getSkillList() {

return skillList;

}

public void setSkillList(Set<Skill> skillList) {

this.skillList = skillList;

}

**Step 2: Update Skill.**

Add the reverse mapping:

@ManyToMany(mappedBy = "skillList")

private Set<Employee> employeeList;

Add getter and setter:

public Set<Employee> getEmployeeList() {

return employeeList;

}

public void setEmployeeList(Set<Employee> employeeList) {

this.employeeList = employeeList;

}

**🔹 Step 3: Modify testGetEmployee() in OrmLearnApplication.**

public static void testGetEmployee() {

Employee employee = employeeService.get(1); // choose valid ID

LOGGER.debug("Employee: {}", employee);

LOGGER.debug("Department: {}", employee.getDepartment());

LOGGER.debug("Skills: {}", employee.getSkillList());

}

This will now work after using fetch = FetchType.EAGER in @ManyToMany.

**🔹 Step 4: Add New Skill to an Employee**

public static void testAddSkillToEmployee() {

Employee employee = employeeService.get(1); // Choose existing employee

Skill skill = skillService.get(3); // Choose skill not yet assigned

Set<Skill> skillList = employee.getSkillList();

skillList.add(skill);

employee.setSkillList(skillList);

employeeService.save(employee);

}

In the main() method:

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

testAddSkillToEmployee();

}

* **One-to-Many:**

@OneToMany(mappedBy = "department", fetch = FetchType.EAGER)

private Set<Employee> employeeList;

* **Many-to-One:**

@ManyToOne

@JoinColumn(name = "dp\_id")

private Department department;

* **Many-to-Many:**

@ManyToMany(fetch = FetchType.EAGER)

@JoinTable(name = "employee\_skill",

joinColumns = @JoinColumn(name = "es\_em\_id"),

inverseJoinColumns = @JoinColumn(name = "es\_sk\_id"))

private Set<Skill> skillList;

**spring Data JPA and Hibernate**

**Exercise 1:** Employee Management System - Overview and Setup

**Business Scenario:**

You are developing an employee management system that will manage employee data, departments, and their relationships.

**Source code:**

Configure application.properties

# H2 Database Configuration

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

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Hibernate Configuration

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

# H2 Console

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

**Exercise 2: Employee Management System - Creating Entities**

**Business Scenario:**

Define JPA entities for Employee and Department with appropriate relationships.

**Source code:**

Department.

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

import lombok.\*;

import .util.List;

@Entity

@Table(name = "department")

@Data // Lombok annotation: includes getters, setters, toString, etc.

@NoArgsConstructor

@AllArgsConstructor

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(nullable = false)

private String name;

// One Department has many Employees

@OneToMany(mappedBy = "department", cascade = CascadeType.ALL, fetch = FetchType.LAZY)

private List<Employee> employees;

}

Employee.

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

import lombok.\*;

@Entity

@Table(name = "employee")

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(nullable = false)

private String name;

@Column(unique = true)

private String email;

// Many employees belong to one department

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "department\_id") // Foreign key column

private Department department;

}

Step 2: Explanation of Relationships

* @OneToMany in Department means one department has many employees.
* @ManyToOne in Employee means each employee belongs to one department.
* mappedBy = "department" means the department field in the Employee class owns the relationship.
* Fetch type is LAZY by default for performance but can be changed to EAGER if needed for testing.

Step 3: Optional Repository Interfaces

In com.example.employeemanagementsystem.repository, create:

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

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

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Department;

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

public interface DepartmentRepository extends JpaRepository<Department, Long> {

}

**Exercise 3: Employee Management System - Creating Repositories**

**Business Scenario:**

Create repositories for Employee and Department entities to perform CRUD operations.

**Source code:**

**Step 1: Your Entity Classes**

**Employee.**

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

private Double salary;

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

// Getters and Setters

}

**Department.**

@Entity

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String location;

// Getters and Setters

}

**Step 2: Create Repository Interfaces**

**EmployeeRepository.**

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Derived query methods

List<Employee> findByName(String name);

List<Employee> findByDepartmentName(String deptName);

List<Employee> findBySalaryGreaterThan(Double salary);

}

**DepartmentRepository.**

public interface DepartmentRepository extends JpaRepository<Department, Long> {

// Derived query methods

Department findByName(String name);

List<Department> findByLocation(String location);

}

**Exercise 4: Employee Management System - Implementing CRUD Operations**

**Business Scenario:** Implement CRUD operations for managing employees and departments.

**Source code:**

EmployeeController.

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

@Autowired

private EmployeeRepository employeeRepository;

// CREATE

@PostMapping

public Employee createEmployee(@RequestBody Employee employee) {

return employeeRepository.save(employee);

}

// READ ALL

@GetMapping

public List<Employee> getAllEmployees() {

return employeeRepository.findAll();

}

// READ BY ID

@GetMapping("/{id}")

public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id) {

return employeeRepository.findById(id)

.map(ResponseEntity::ok)

.orElse(ResponseEntity.notFound().build());

}

// UPDATE

@PutMapping("/{id}")

public ResponseEntity<Employee> updateEmployee(@PathVariable Long id, @RequestBody Employee updatedEmployee) {

return employeeRepository.findById(id)

.map(employee -> {

employee.setName(updatedEmployee.getName());

employee.setEmail(updatedEmployee.getEmail());

employee.setSalary(updatedEmployee.getSalary());

employee.setDepartment(updatedEmployee.getDepartment());

return ResponseEntity.ok(employeeRepository.save(employee));

}).orElse(ResponseEntity.notFound().build());

}

// DELETE

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteEmployee(@PathVariable Long id) {

if (employeeRepository.existsById(id)) {

employeeRepository.deleteById(id);

return ResponseEntity.noContent().build();

} else {

return ResponseEntity.notFound().build();

}

}

}

DepartmentController.

@RestController

@RequestMapping("/api/departments")

public class DepartmentController {

@Autowired

private DepartmentRepository departmentRepository;

// CREATE

@PostMapping

public Department createDepartment(@RequestBody Department department) {

return departmentRepository.save(department);

}

// READ ALL

@GetMapping

public List<Department> getAllDepartments() {

return departmentRepository.findAll();

}

// READ BY ID

@GetMapping("/{id}")

public ResponseEntity<Department> getDepartmentById(@PathVariable Long id) {

return departmentRepository.findById(id)

.map(ResponseEntity::ok)

.orElse(ResponseEntity.notFound().build());

}

// UPDATE

@PutMapping("/{id}")

public ResponseEntity<Department> updateDepartment(@PathVariable Long id, @RequestBody Department updatedDept) {

return departmentRepository.findById(id)

.map(department -> {

department.setName(updatedDept.getName());

department.setLocation(updatedDept.getLocation());

return ResponseEntity.ok(departmentRepository.save(department));

}).orElse(ResponseEntity.notFound().build());

}

// DELETE

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteDepartment(@PathVariable Long id) {

if (departmentRepository.existsById(id)) {

departmentRepository.deleteById(id);

return ResponseEntity.noContent().build();

} else {

return ResponseEntity.notFound().build();

}

}

}

**Exercise 5: Employee Management System - Defining Query Methods**

**Business Scenario:**

Enhance your repository to support custom queries.

**Source code:**

**Defining Query Methods**

Spring Data JPA allows you to define custom query methods in two main ways:

**A. Using Keywords in Method Names (Derived Queries)**

**In EmployeeRepository.**

// Find employees by name

List<Employee> findByName(String name);

// Find employees by department name

List<Employee> findByDepartmentName(String deptName);

// Find employees with salary greater than a certain amount

List<Employee> findBySalaryGreaterThan(Double salary);

// Find employees whose name starts with a given prefix

List<Employee> findByNameStartingWith(String prefix);

**B. Using the @Query Annotation (Custom JPQL Queries)**

**In EmployeeRepository.**

@Query("SELECT e FROM Employee e WHERE e.salary > :minSalary")

List<Employee> getEmployeesWithSalaryGreaterThan(@Param("minSalary") Double minSalary);

@Query("SELECT e FROM Employee e WHERE e.department.name = :deptName")

List<Employee> getEmployeesByDepartmentName(@Param("deptName") String deptName);

**In DepartmentRepository.**

@Query("SELECT d FROM Department d WHERE d.location = :location")

List<Department> getDepartmentsByLocation(@Param("location") String location);

**2. Named Queries**

**A. Define Named Queries in the Entity Class**

**➤ In Employee.**

@Entity

@NamedQueries({

@NamedQuery(

name = "Employee.findByEmail",

query = "SELECT e FROM Employee e WHERE e.email = :email"

),

@NamedQuery(

name = "Employee.findHighEarners",

query = "SELECT e FROM Employee e WHERE e.salary >= :salary"

)

})

public class Employee {

// fields, getters, setters

}

**B. Use Named Queries in Repository**

**In EmployeeRepository.**

@Query(name = "Employee.findByEmail")

Employee findByEmail(@Param("email") String email);

@Query(name = "Employee.findHighEarners")

List<Employee> findHighEarners(@Param("salary") Double salary);

**Exercise 6: Employee Management System - Implementing Pagination and Sorting**

**Business Scenario:**

Add pagination and sorting capabilities to your employee search functionality.

**Pagination Using Page and Pageable**

**In EmployeeRepository.**

You **don’t need to add any extra method** to support pagination — JpaRepository already has it:

Page<Employee> findAll(Pageable pageable);

**2. Implement Pagination + Sorting in the Controller**

**In EmployeeController.**

@GetMapping("/paged")

public Page<Employee> getPagedEmployees(

@RequestParam(defaultValue = "0") int page,

@RequestParam(defaultValue = "5") int size,

@RequestParam(defaultValue = "id") String sortBy,

@RequestParam(defaultValue = "asc") String direction) {

Sort sort = direction.equalsIgnoreCase("asc") ?

Sort.by(sortBy).ascending() :

Sort.by(sortBy).descending();

Pageable pageable = PageRequest.of(page, size, sort);

return employeeRepository.findAll(pageable);

}

**3. Sample API Calls**

| **Use Case** | **URL** |
| --- | --- |
| Get first 5 employees sorted by name ascending | /api/employees/paged?page=0&size=5&sortBy=name&direction=asc |
| Get page 2 of size 3 sorted by salary descending | /api/employees/paged?page=1&size=3&sortBy=salary&direction=desc |

**Optional: Custom Pageable Queries**

You can also paginate **filtered results**:

**🔹 In EmployeeRepository.**

Page<Employee> findByDepartmentName(String deptName, Pageable pageable);

**🔹 In EmployeeController.**

@GetMapping("/paged/by-department")

public Page<Employee> getEmployeesByDepartmentPaged(

@RequestParam String deptName,

@RequestParam(defaultValue = "0") int page,

@RequestParam(defaultValue = "5") int size) {

Pageable pageable = PageRequest.of(page, size);

return employeeRepository.findByDepartmentName(deptName, pageable);

}

**Exercise 7: Employee Management System - Enabling Entity Auditing**

**Business Scenario:**

Implement auditing to track the creation and modification of employees and departments.

**Source code:**

**Step 1: Enable Auditing in Spring Boot**

**🔹 In your main class (e.g., EmployeeManagementApplication. )**

@SpringBootApplication

@EnableJpaAuditing(auditorAwareRef = "auditorProvider")

public class EmployeeManagementApplication {

public static void main(String[] args) {

SpringApplication.run(EmployeeManagementApplication.class, args);

}

}

**Step 2: Create AuditorAware Implementation**

This component provides the current user (or system) name.

**🔹 AuditorAwareImpl.**

import org.springframework.data.domain.AuditorAware;

import org.springframework.stereotype.Component;

import .util.Optional;

@Component("auditorProvider")

public class AuditorAwareImpl implements AuditorAware<String> {

@Override

public Optional<String> getCurrentAuditor() {

// Replace this with Spring Security user if available

return Optional.of("SYSTEM\_USER");

}

}

**Step 3: Update Entity Classes with Auditing Fields**

**🔹 Employee.**

import org.springframework.data.annotation.CreatedBy;

import org.springframework.data.annotation.LastModifiedBy;

import org.springframework.data.annotation.CreatedDate;

import org.springframework.data.annotation.LastModifiedDate;

import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import x.persistence.\*;

import .time.LocalDateTime;

@Entity

@EntityListeners(AuditingEntityListener.class)

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

private Double salary;

@ManyToOne

private Department department;

@CreatedDate

private LocalDateTime createdDate;

@LastModifiedDate

private LocalDateTime lastModifiedDate;

@CreatedBy

private String createdBy;

@LastModifiedBy

private String lastModifiedBy;

// Getters & Setters

}

**🔹 Department.**

@Entity

@EntityListeners(AuditingEntityListener.class)

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String location;

@CreatedDate

private LocalDateTime createdDate;

@LastModifiedDate

private LocalDateTime lastModifiedDate;

@CreatedBy

private String createdBy;

@LastModifiedBy

private String lastModifiedBy;

// Getters & Setters

}

**Step 4: Add JPA Auditing Dependencies (if missing)**

In pom.xml:

xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

These annotations work only if you are using Spring Boot with Spring Data JPA and have the auditing enabled.

**Exercise 8: Employee Management System - Creating Projections**

**Business Scenario:**

Create projections to fetch specific data subsets from the employee and department entities.

**Source code:**

**Interface-Based Projections**

**🔹 Define a Projection Interface**

public interface EmployeeNameAndEmail {

String getName();

String getEmail();

}

**🔹 Use It in the Repository**

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<EmployeeNameAndEmail> findByDepartmentName(String departmentName);

}

**Class-Based Projections (DTO Projections)**

**🔹 Define a DTO Class**

public class EmployeeDTO {

private String name;

private Double salary;

public EmployeeDTO(String name, Double salary) {

this.name = name;

this.salary = salary;

}

// Getters

}

**🔹 Use Constructor Expression with @Query**

@Query("SELECT new com.example.dto.EmployeeDTO(e.name, e.salary) FROM Employee e WHERE e.salary > :minSalary")

List<EmployeeDTO> findEmployeesWithSalaryGreaterThan(@Param("minSalary") double minSalary);