**WEEK-3**

**SPRING DATA JPA AND HIBERNATE HANDSON**

### **Why ORM (Object-Relational Mapping) matters, and when it doesn’t**

**ORM bridges the gap between object-oriented code, like Java classes, and relational databases, like MySQL or PostgreSQL. Instead of writing raw SQL, you work with objects, and the ORM takes care of the mapping behind the scenes.**

**What makes ORM useful:  
Cuts down on repetitive boilerplate, no more JDBC setup every time,  
automatically maps tables to classes and rows to objects, making code more readable and less error-prone,  
makes your app less dependent on a specific database, so switching vendors becomes easier,  
comes with useful features like caching, lazy loading, and transaction handling,  
helps prevent SQL injection by using safe, parameterized queries.**

**Disadvantages:  
Can affect performance if not optimized properly,  
limits your control if you rely only on ORM and ignore SQL’s full capabilities,  
for complex queries, you’ll often fall back to native SQL or JPQL.**

**2. Need and Benefits of Spring Data JPA**

Spring Data JPA is part of the Spring ecosystem that builds on top of JPA and Hibernate. It simplifies building the data access layer in Spring applications.

**How it evolved:**  
 Hibernate started with XML-based configuration to map classes to database tables,then came annotations like @Entity, @Id, etc., which made configuration easier, Spring Data JPA takes it further by removing the need to write implementation code for common queries.

**Key benefits:**  
Reduces boilerplate code with built-in interfaces like JpaRepository,  
supports CRUD operations and custom queries just by using method names,  
integrates seamlessly with Spring Boot and other Spring modules,  
supports in-memory databases like H2 for fast development and testing.

**3. Core Objects of Hibernate Framework**

Hibernate is the most widely used ORM implementation in Java, and it works through a few core components:

**SessionFactory**  
A thread-safe factory that creates Session objects, created once and reused across the application,

**Session**  
A lightweight, single-threaded object used for database interactions like CRUD operations,

**Transaction**  
Handles commit and rollback, ensures data consistency during operations,

**Connection Provider**  
Manages JDBC connections for Hibernate behind the scenes,

**TransactionFactory**  
A strategy interface used to create transaction instances depending on the underlying environment.

These components work together to separate database logic from business logic, making the codebase cleaner and easier to manage.

**4. ORM Implementation Using Hibernate XML and Annotation Configuration**

Hibernate can be configured in two primary ways:

**XML Configuration**  
Define your entity class, like Employee.java,  
create a separate XML file (Employee.hbm.xml) to map class fields to table columns,  
configure database settings and mappings in hibernate.cfg.xml,  
load the configuration, get a SessionFactory, and manage sessions and transactions.

**Annotation Configuration**  
Use annotations like @Entity, @Table, @Id, and @Column directly in the class,  
skip the XML mapping file, but still use hibernate.cfg.xml for database settings,  
this approach is more modern and commonly used in real-world projects.

**Hands on 1**

**Spring Data JPA - Quick Example**   
  
**Software Pre-requisites**

* MySQL Server 8.0
* MySQL Workbench 8
* Eclipse IDE for Enterprise Java Developers 2019-03 R
* Maven 3.6.2

**Create a Eclipse Project using Spring Initializr**

> mysql -u root -p

mysql> create schema ormlearn;

cc

Country table creation

* Create a new table country with columns for code and name. For sample, let us insert one country with values 'IN' and 'India' in this table.

create table country(co\_code varchar(2) primary key, co\_name varchar(50));

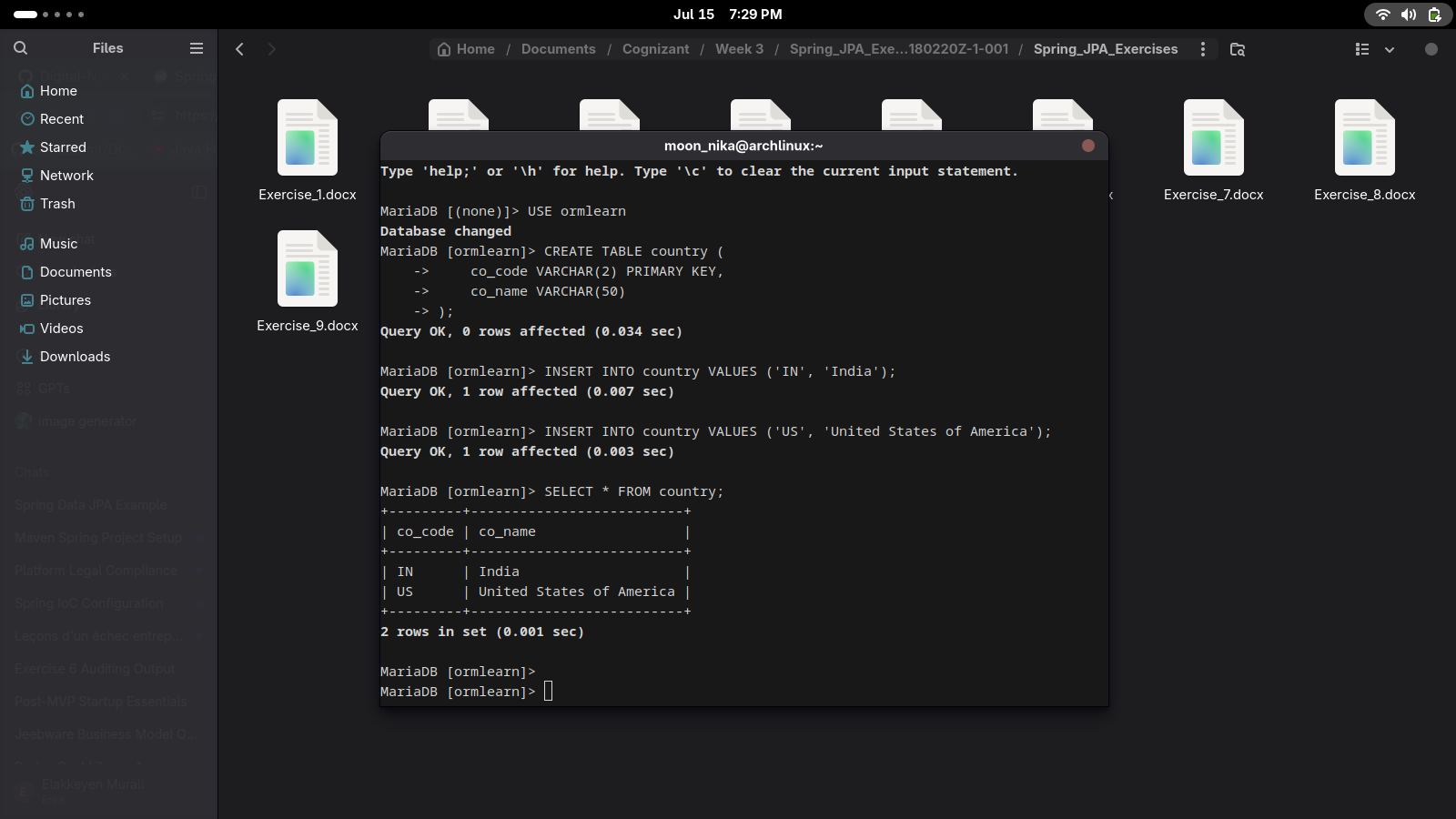
* Insert couple of records into the table

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

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

insert into country values(‘JP’,’JAPAN’);

Creation:



**Persistence Class - com.cognizant.orm-learn.model.Country**

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 and setters

  // toString()

}

**Repository Class - com.cognizant.orm-learn.CountryRepository**

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> {

}

**Service Class - com.cognizant.orm-learn.service.CountryService  
Testing in OrmLearnApplication.java**

package com.cognizant.orm\_learn;

import java.util.List;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import com.cognizant.orm\_learn.model.Country;

import com.cognizant.orm\_learn.service.CountryService;

@SpringBootApplication

public class OrmLearnApplication {

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

private static CountryService countryService;

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(OrmLearnApplication.class, args);

countryService = context.getBean(CountryService.class);

LOGGER.info("Inside main");

testGetAllCountries();

}

private static void testGetAllCountries() {

LOGGER.info("Start");

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

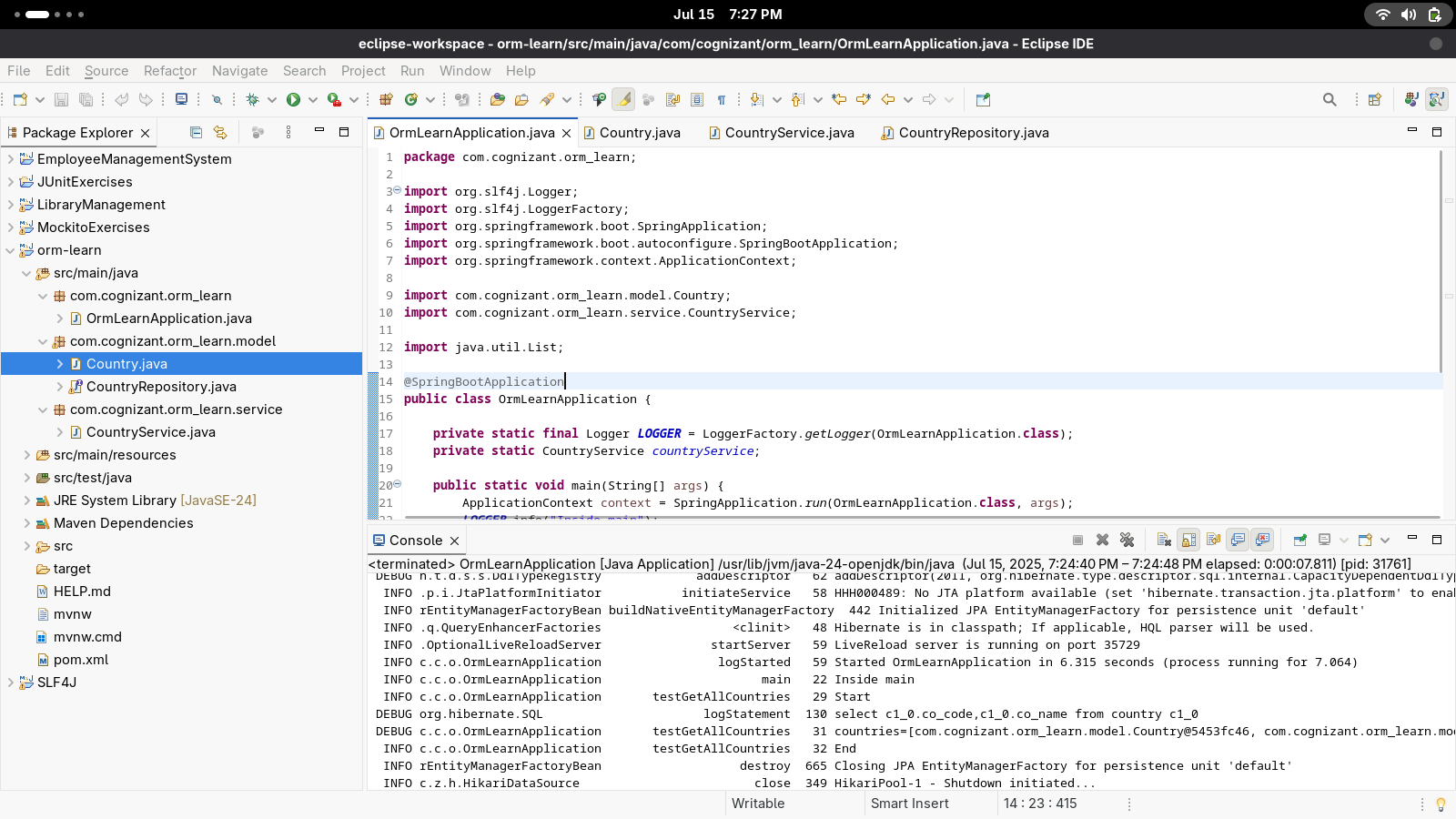
LOGGER.debug("countries={}", countries);

LOGGER.info("End");

}

}

**OUTPUT:**

****

**HANDSON 4**

**5. Difference between JPA, Hibernate, and Spring Data JPA**

**JPA**  
Think of JPA as the blueprint, it’s just a specification,  
it defines what should be done for object-relational mapping, but not how,  
there’s no working code, only guidelines and interfaces.

**Hibernate**  
Hibernate brings JPA to life, it’s one of the most widely used implementations,  
offers extra features like caching, lazy loading, and HQL,can be used with or without JPA.

**Spring Data JPA**  
Builds on top of JPA and Hibernate, and makes data access even easier,  
lets you define interfaces like UserRepository and use method names like findByEmail(),no need to write query code manually.

**6. DML Operations using Spring Data JPA (on a single table)**

Spring Data JPA simplifies basic CRUD and DML operations on entities.

**Setup:**

* Add Spring Boot, JPA, and H2/MySQL dependencies.
* Create your entity class with annotations (@Entity, @Id, etc.).
* Create a repository interface extending JpaRepository<Entity, ID>.

**DML Examples:**

* findById(id) → fetch a single record.
* save(entity) → insert or update a record.
* deleteById(id) → delete a record.
* findAll() → fetch all records.
* Custom queries like findByNameContaining(String name) → use query methods.

Also, configure application.properties:

properties

CopyEdit

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

This helps with logging and schema generation.