**WEEK-3**

**SPRING DATA JPA AND HIBERNATE HANDSON**

**1. Need and Benefits of ORM (Object-Relational Mapping)**

ORM (Object-Relational Mapping) is a technique used to map Java (or other language) objects to database tables. Instead of writing raw SQL queries, developers interact with the database using objects and classes. This makes the code easier to understand and maintain.

**Benefits of ORM:**

* Reduces boilerplate SQL code and JDBC handling.
* Simplifies database interactions with automatic mapping between classes and tables.
* Makes the application database-agnostic (e.g., switching from MySQL to PostgreSQL becomes easier).
* Supports features like lazy loading, caching, and transaction management.
* Helps avoid SQL injection through parameterized queries.

**Drawbacks:**

* Slight performance overhead in some scenarios.
* May hide the power of SQL if not used carefully.
* Complex queries might still need native SQL or JPQL.

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

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

**Evolution:**

* Initially, Hibernate used XML-based configuration to map classes to database tables.
* Then annotations (like @Entity, @Id, etc.) made configuration easier.
* Spring Data JPA takes it further by eliminating the need to write implementation code for common queries.

**Benefits of Spring Data JPA:**

* Reduces boilerplate code with built-in repository interfaces like JpaRepository.
* Easily supports CRUD operations and custom queries using method names.
* Integrates smoothly 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 popular ORM implementation in Java, and it relies on a few key components:

* **SessionFactory**: A thread-safe factory that creates Session objects. It's created once and reused.
* **Session**: A lightweight, single-threaded object used to interact with the database (CRUD operations).
* **Transaction**: Handles commit and rollback operations.
* **Connection Provider**: Manages JDBC connections for Hibernate.
* **TransactionFactory**: Strategy interface for creating transaction instances.

These components work together to abstract the database interaction layer from the business logic.

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

There are two main ways to configure Hibernate:

**XML Configuration:**

* Define your entity class (e.g., Employee.java).
* Create an XML file (Employee.hbm.xml) to map class fields to table columns.
* Configure Hibernate settings in hibernate.cfg.xml.
* Load the configuration, get SessionFactory, and manage sessions and transactions.

**Annotation Configuration:**

* Use annotations like @Entity, @Table, @Id, and @Column in the class directly.
* Skip mapping XML, but still use hibernate.cfg.xml for DB settings.
* This is more modern and widely used.

Both approaches follow a similar flow: load config → open session → begin transaction → persist data → commit → close session.

**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));

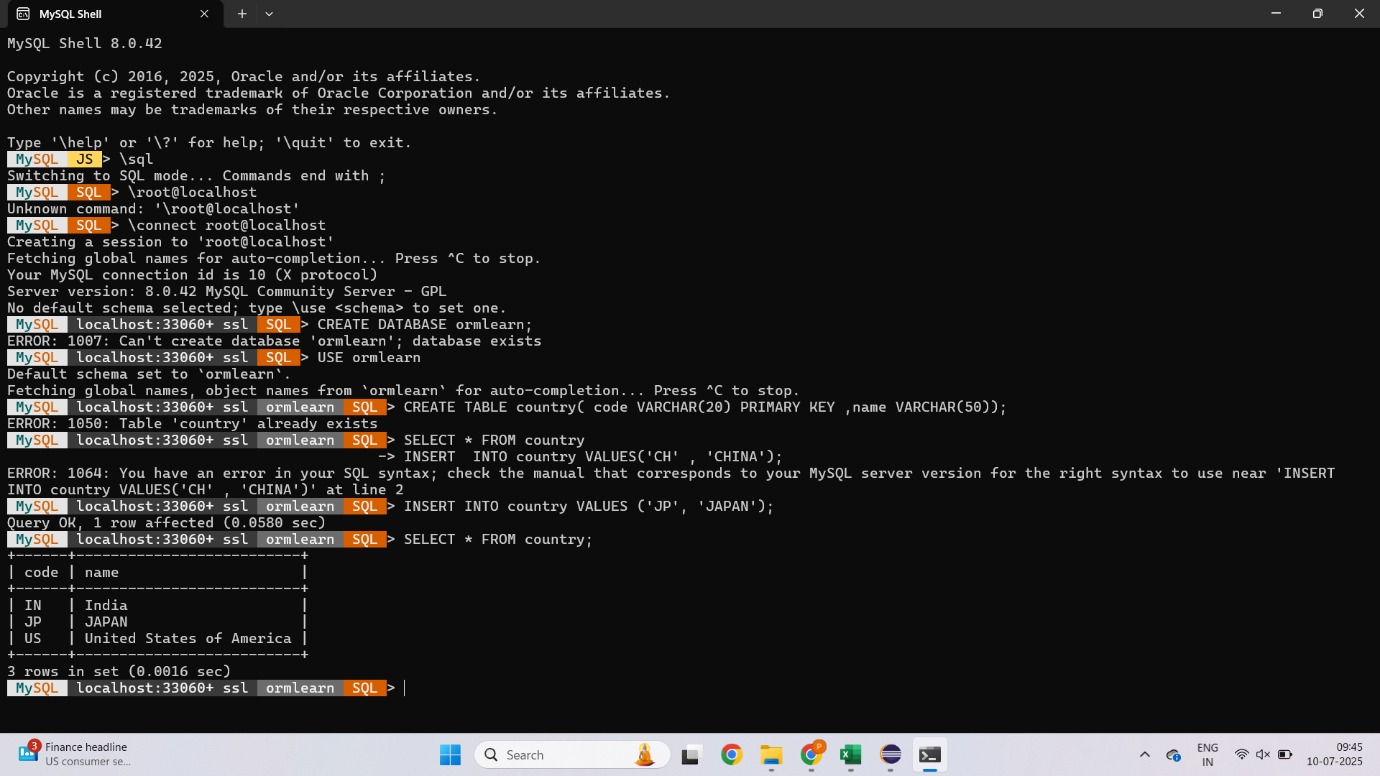
* Insert couple of records into the table

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();

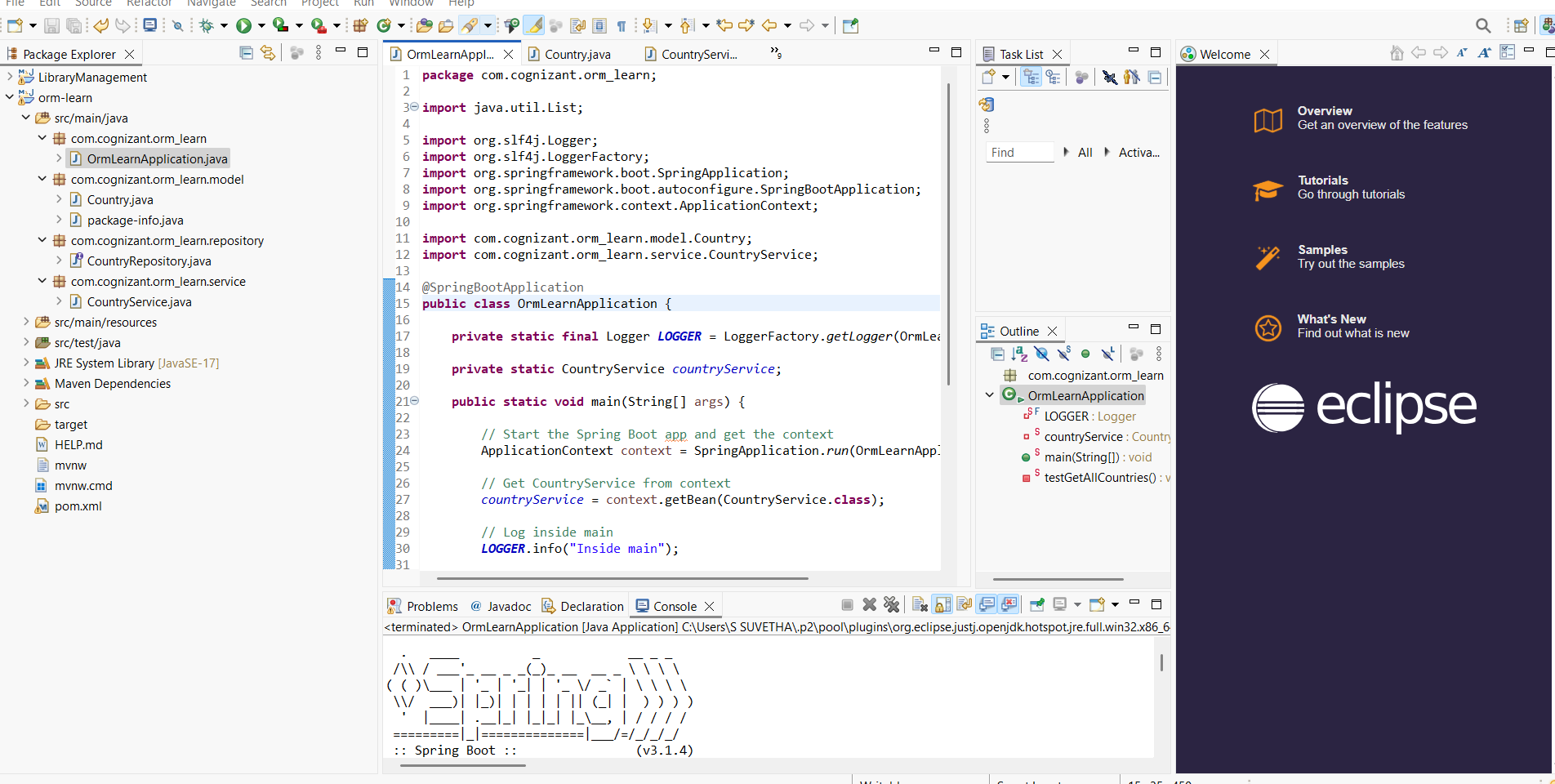
LOGGER.debug("countries={}", countries);

LOGGER.info("End");

}

}

**OUTPUT:**



**HANDSON 4**

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

* **JPA**: Java Persistence API — a specification (just interfaces and rules). It defines how Java objects should be mapped to database tables, but it doesn't provide actual implementation.
* **Hibernate**: A popular implementation of JPA. It adds many additional features beyond JPA and can be used on its own.
* **Spring Data JPA**: A higher-level abstraction built on top of JPA and Hibernate. It reduces even more boilerplate by allowing developers to define repository interfaces with method names like findByName() without writing query code.

**In short:**

* JPA is the *standard*,
* Hibernate is an *implementation*,
* Spring Data JPA is an *abstraction layer* to make JPA+Hibernate easier to use.

**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.