**Spring Data JPA Handson**

**Hands On 1 : Create a Eclipse Project using Spring Initializr**

**Program:**

**Country.java**

package com.cognizant.orm\_learn.model;

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

@Entity

public class Country {

@Id

private String code;

private String name;

// Getters and Setters

public String getCode() {

return code;

}

public void setCode(String code) {

this.code = code;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

**CountryRepository.java**

package com.cognizant.orm\_learn.repository;

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

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

public interface CountryRepository extends JpaRepository<Country, String> {

}

**CountryService.java**

package com.cognizant.orm\_learn.service;

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.orm\_learn.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();

}

}

**Application.properties:**

spring.application.name=orm-learn

# Logging

logging.level.org.springframework=info

logging.level.com.cognizant=debug

logging.level.org.hibernate.SQL=trace

logging.level.org.hibernate.type.descriptor.sql=trace

logging.pattern.console=%d{dd-MM-yy} %d{HH:mm:ss.SSS} %-20.20thread %5p %-25.25logger**{25}** %25M %4L %m%n

# DB Config

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/ormlearn

spring.datasource.username=root

spring.datasource.password=Poorani123@

# Hibernate

spring.jpa.hibernate.ddl-auto=validate

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

**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 CountryService *countryService*;

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

public static void main(String[] args) {

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

*LOGGER*.info("Inside main");

*countryService* = context.getBean(CountryService.class);

*testGetAllCountries*();

}

private static void testGetAllCountries() {

*LOGGER*.info("Start");

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

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

*LOGGER*.info("End");

}

}

**Creating a database with Country table and inserting values:**

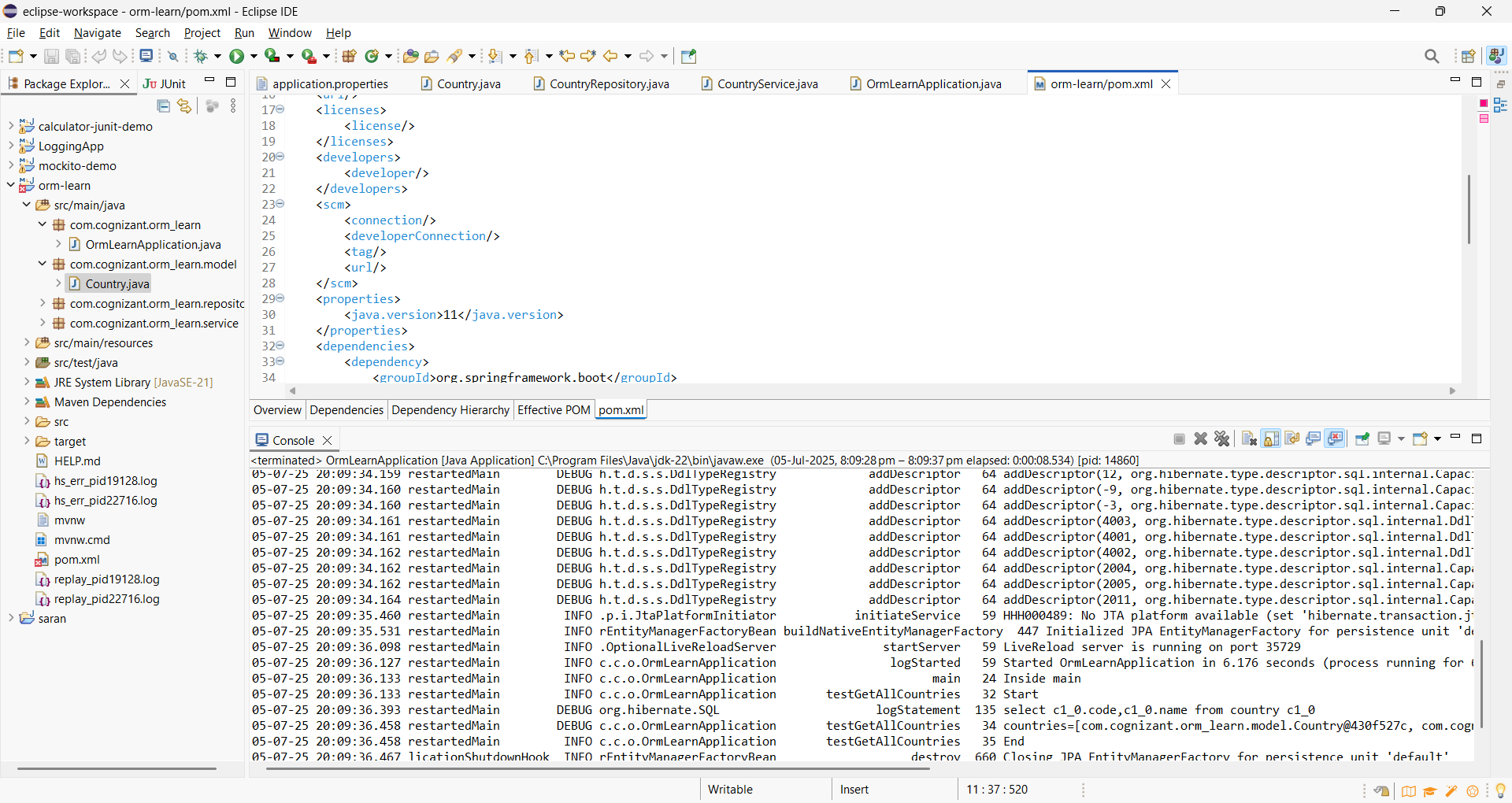
create schema ormlearn;

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

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

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

**Output:**



**Hands on 2 : Hibernate XML Config implementation walk through**

1. **Explain how object to relational database mapping done in hibernate xml configuration file**

* Hibernate uses XML files (like hibernate.cfg.xml and mapping files .hbm.xml) to map Java classes (objects) to database tables (relational).
* hibernate.cfg.xml configures connection details (DB URL, username, password, dialect, driver).
* Mapping files (\*.hbm.xml) describe how Java class properties map to database table columns.

**Example of Mapping XML (Country.hbm.xml):**

<class name="com.example.Country" table="COUNTRY">

<id name="code" column="COUNTRY\_CODE"/>

<property name="name" column="COUNTRY\_NAME"/>

</class>

1. **Explain about following aspects of implementing the end to end operations in Hibernate:**
   * **SessionFactory**

* It is a factory for Session objects.
* Created once during application startup.
* It is thread-safe and heavyweight, so you should only have one per database.
* It is created using the Configuration object by reading hibernate.cfg.xml.

Example:

SessionFactory factory = new Configuration().configure().buildSessionFactory();

* + **Session**
* A Session represents a **single-threaded unit of work** with the database.
* It is used to create, read, update, and delete persistent objects.
* Sessions are **not thread-safe** and should be closed after use.

Example:

Session session = factory.openSession();

* + **Transaction**
* A transaction represents a **unit of work** in the database.
* Helps ensure **data consistency** by allowing commit or rollback.
* It is used to **group multiple operations** into a single atomic unit.

Example:

Transaction tx = session.beginTransaction();

* + **beginTransaction()**
* Begins a new database transaction.
* Should always be called before performing any write operation (save, update, delete).

Example:

Transaction tx = session.beginTransaction();

* + **commit()**
* Commits the current transaction to the database.
* Saves all changes made during the session.

Example:

tx.commit();

* + **rollback()**
* Rolls back the current transaction if any error occurs.
* Ensures that no partial or invalid data is stored.

Example:

tx.rollback();

* + **session.save()**
* Saves a Java object (entity) to the database.
* The object is inserted as a new row in the mapped table.

Example:

Country country = new Country("IN", "India");

session.save(country);

* + **session.createQuery().list()**
* Executes a **Hibernate Query Language (HQL)** query and returns a list of results.
* It is used for **select operations** involving multiple records.

Example:

List<Country> countries = session.createQuery("from Country").list();

* + **session.get()**
* Retrieves an object from the database using its primary key.
* Returns null if no matching row is found.

Example:

Country country = session.get(Country.class, "IN");

* + **session.delete()**
* Deletes a record (row) from the database that corresponds to the given object.

Example:

session.delete(country);

**Hands on 3 : Hibernate Annotation Config implementation walk through**

* **Explain how object to relational database mapping done in persistence class file Employee**

In the annotation-based approach, the Employee class uses **JPA annotations** to define mappings between class members and database table columns.

**Example:**

@Entity

@Table(name = "EMPLOYEE")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private int id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column(name = "salary")

private int salary;

// Getters and Setters

}

This class tells Hibernate how to persist Employee objects in the EMPLOYEE table.

* **Explain about following aspects of implementing the end to end operations in Hibernate:**
  + **@Entity**
* Marks the class as a **persistent entity** (a table will be mapped to this class).
* Required for Hibernate to recognize the class as a database-mapped object.

**Example:**

@Entity

* + **@Table**
* Specifies the name of the **table** in the database that this entity maps to.
* If omitted, the table name defaults to the class name.

**Example:**

@Table(name = "EMPLOYEE")

* + **@Id**
* Denotes the **primary key** of the entity.
* Required for every entity to uniquely identify rows.

**Example:**

@Id

* + **@GeneratedValue**
* Specifies the **auto-generation strategy** for the primary key.
* Common strategies: AUTO, IDENTITY, SEQUENCE, TABLE.

**Example:**

@GeneratedValue(strategy = GenerationType.IDENTITY)

* + **@Column**
* Maps a Java field to a specific **column** in the table.
* You can define name, length, nullable, unique, etc.

**Example:**

@Column(name = "first\_name")

* + **Hibernate Configuration (hibernate.cfg.xml)**

The hibernate.cfg.xml is used to configure Hibernate runtime settings like database connection, dialect, and mapping resources.

**Example:**

<hibernate-configuration>

<session-factory>

<!-- JDBC Configuration -->

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/your\_db</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">password</property>

<!-- Hibernate Properties -->

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="hibernate.hbm2ddl.auto">update</property>

<property name="show\_sql">true</property>

<!-- Mapping Class -->

<mapping class="com.example.Employee"/>

</session-factory>

</hibernate-configuration>

* + - **Dialect**
* Specifies the SQL dialect Hibernate should use.
* Example: org.hibernate.dialect.MySQLDialect, OracleDialect, etc.
* This helps Hibernate generate database-specific SQL.

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

* + - **Driver**
* The fully qualified Java class name of the **JDBC driver**.

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

* + - **Connection URL**
* JDBC URL to connect to the database.

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/your\_db</property>

* + - **Username**
* The **username** for the database.

<property name="hibernate.connection.username">root</property>

* + - **Password**
* The password for the database user.

<property name="hibernate.connection.password">password</property>

Annotation-based configuration in Hibernate simplifies mapping and reduces the need for separate XML files. The @Entity, @Table, @Id, and other annotations directly link Java classes and fields to their corresponding database tables and columns. Together with hibernate.cfg.xml, this allows seamless object-relational integration for modern Java applications.