**Hands on 1**

**Spring Data JPA - Quick Example**

**application.properties**

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=root

spring.jpa.hibernate.ddl-auto=validate

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

logging.level.org.springframework=info

logging.level.com.cognizant=debug

logging.level.org.hibernate.SQL=trace

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

**OrmLearnApplication.java**

package com.cognizant.ormlearn;

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.ormlearn.model.Country;

import com.cognizant.ormlearn.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);

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");

}

}

**Country.java**

package com.cognizant.ormlearn.model;

import javax.persistence.\*;

@Entity

@Table(name = "country")

public class Country {

@Id

@Column(name = "code")

private String code;

@Column(name = "name")

private String name;

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;

}

@Override

public String toString() {

return "Country [code=" + code + ", name=" + name + "]";

}

}

**CountryRepository.java**

package com.cognizant.ormlearn.repository;

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> {

}

**CountryService.java**

package com.cognizant.ormlearn.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.ormlearn.model.Country;

import com.cognizant.ormlearn.repository.CountryRepository;

@Service

public class CountryService {

@Autowired

private CountryRepository countryRepository;

@Transactional

public List<Country> getAllCountries() {

return countryRepository.findAll();

}

}

**SQL (MySQL Workbench)**

create database ormlearn;

use ormlearn;

create table country (

code varchar(2) primary key,

name varchar(50)

);

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

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

**Hands on 4**

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

**Java Persistence API (JPA)**

The Java Persistence API (JPA) is a formal specification (JSR 338) that defines a standardized way to map Java objects to relational database tables, manage entity lifecycles, and perform queries using JPQL or the Criteria API. As a pure interface and annotation model, JPA itself provides no runtime behaviour, it simply prescribes how persistence should work in a provider‑independent fashion, ensuring that code written against JPA interfaces can switch implementations without significant rewrites.

// EntityManager usage under pure JPA

EntityManagerFactory emf = Persistence.createEntityManagerFactory("my-pu");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

em.persist(new Employee("Alice", "Engineering"));

em.getTransaction().commit();

em.close();

emf.close();

**Hibernate**

Hibernate is a full‑featured object‑relational mapping (ORM) framework that serves as one of the most popular JPA implementations. It not only fully honors the JPA contracts (via its EntityManager façade) but also offers its own native API (Session), multi‑level caching, performance‑tuning tools, and proprietary extensions. Those who require advanced caching strategies, custom database dialects, or fine‑grained control often turn to Hibernate’s additional capabilities beyond the baseline JPA specification.

// Native Hibernate Session usage

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

Session session = factory.openSession();

session.beginTransaction();

Integer id = (Integer) session.save(new Employee("Bob", "Marketing"));

session.getTransaction().commit();

session.close();

factory.close();

**Spring Data JPA**

Spring Data JPA is a Spring‑managed abstraction layer that sits atop any JPA provider (commonly Hibernate) to drastically reduce boilerplate code. By defining simple repository interfaces—extending JpaRepository—developers gain auto‑generated implementations for CRUD operations, pagination, sorting, and query derivation based on method names. Spring Data JPA also leverages Spring’s declarative transaction management, allowing persistence operations to execute within well‑defined transactional boundaries without explicit EntityManager or Session handling.

// 1. Repository interface

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

// 2. Service using the repository

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository repo;

@Transactional

public Employee addEmployee(Employee employee) {

return repo.save(employee);

}

}