**1.Spring Data JPA - Quick Example**

**Answer:**

**Program:**

//Country.java – Model

package com.cognizant.orm\_learn.model;

import jakarta.persistence.Entity;

import jakarta.persistence.Table;

import jakarta.persistence.Column;

import jakarta.persistence.Id;

@Entity

@Table(name="country")

public class Country {

    @Id

    @Column(name="co\_code")

    private String code;

    @Column(name="co\_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 String.format("Code : %s Name : %s", code, name);

    }

}

//CountryRepository.java – Repository

package com.cognizant.orm\_learn.repository;

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

import org.springframework.stereotype.Repository;

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

@Repository

public interface CountryRepository extends JpaRepository<Country, String> {

}

//CountryService.java – Service

package com.cognizant.orm\_learn.service;

import org.springframework.stereotype.Service;

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

import java.util.List;

import com.cognizant.orm\_learn.repository.CountryRepository;

import jakarta.transaction.Transactional;

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

@Service

public class CountryService {

    private final CountryRepository countryRepository;

    public CountryService( @Autowired CountryRepository countryRepository) {

        this.countryRepository = countryRepository;

    }

    @Transactional

    public List<Country> getAllCountry() {

        return countryRepository.findAll();

    }

}

//OrmLearnApplication.java

package com.cognizant.orm\_learn;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.slf4j.LoggerFactory;

import org.slf4j.Logger;

import org.springframework.context.ApplicationContext;

import java.util.List;

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

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

@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();

    }

    public static void testGetAllCountries() {

        LOGGER.info("Start");

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

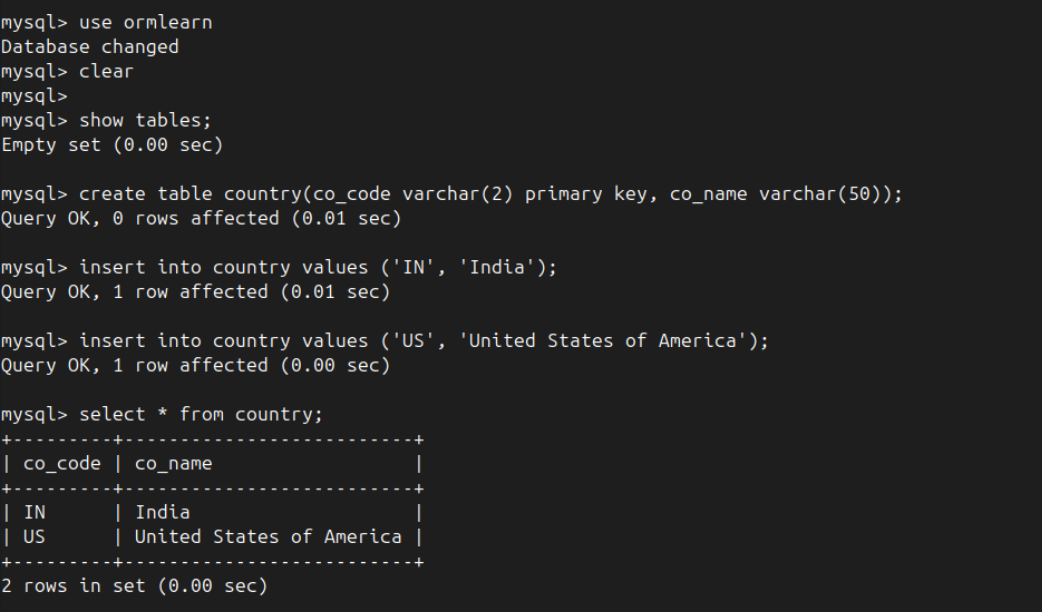
        LOGGER.debug("countries={}", countries);

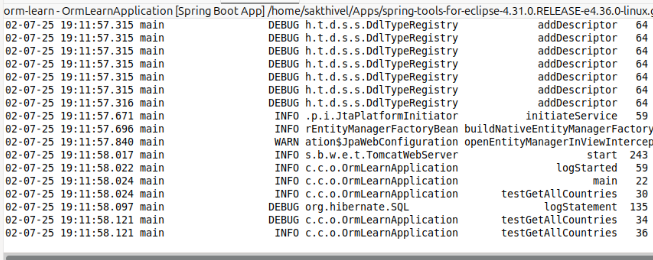
        LOGGER.info("End");

    }

}

**Output:**





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

**1. Java Persistence API (JPA)**

**JPA** is a **Java specification** that defines how Java objects should be mapped to relational database tables. It’s essentially a set of interfaces and annotations that allow developers to interact with databases in an object-oriented way.

* **Nature**: It's just a **contract**, not an implementation.
* **Key Components**:
  + @Entity – Declares a class as a persistent entity.
  + @Id – Marks a field as the primary key.
  + EntityManager – The interface used to manage database operations.

JPA **doesn’t provide the actual code** to perform database operations. Instead, it **requires a provider** like Hibernate or EclipseLink to implement its behavior.

**2. Hibernate**

**Hibernate** is a **popular ORM (Object Relational Mapping) framework** that provides the concrete implementation of the JPA specification. It translates Java objects into database rows and vice versa.

* **Implements JPA**: Yes, fully. But it also offers **extra features** beyond JPA, such as:
  + First-level and second-level caching
  + Lazy/eager loading strategies
  + Batch processing and performance tuning options
* **Manual Management**: When using Hibernate directly, the developer is responsible for managing sessions, transactions, and exception handling.

Hibernate can be used **with or without JPA annotations** — you can use its native XML or annotation-based configurations.

**3. Spring Data JPA**

**Spring Data JPA** is a **part of the Spring ecosystem** that simplifies JPA-based data access. It builds on top of JPA and adds an abstraction layer that minimizes boilerplate code for repositories and queries.

* **Not a JPA implementation** – It still depends on a provider like Hibernate underneath.
* **Reduces Code**: You don’t need to write common CRUD operations; you just declare interfaces.
* **Features**:
  + Auto-implementation of basic repository methods (save(), findAll(), deleteById()).
  + Support for custom queries via method naming or @Query annotation.
  + Integrated transaction management using Spring’s @Transactional.

It works seamlessly with Spring Boot and Spring’s dependency injection model, making development much faster and cleaner.

**Code Comparison: Hibernate vs Spring Data JPA**

**Using Hibernate :**

public Integer addEmployee(Employee employee) {

    Session session = factory.openSession();

    Transaction tx = null;

    Integer employeeId = null;

    try {

        tx = session.beginTransaction();

        employeeId = (Integer) session.save(employee); // Save

        tx.commit(); // Commit

    } catch (Exception e) {

        if (tx != null) tx.rollback(); // Rollback

        e.printStackTrace();

    } finally {

        session.close();

    }

    return employeeId;

}

**Using Spring Data JPA:**

//EmployeeRepository.java

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

    // No need to define save(), findById(), deleteById(), etc.

}

//EmployeeService.java

@Service

public class EmployeeService {

    @Autowired

    private EmployeeRepository employeeRepository;

    @Transactional

    public void addEmployee(Employee employee) {

        employeeRepository.save(employee); // Done in one line

    }

}

**Differences**

|  |  |  |  |
| --- | --- | --- | --- |
| **Aspect** | **JPA** | **Hibernate** | **Spring Data JPA** |
| Type | Specification (API) | ORM framework | Spring abstraction over JPA |
| Provides Implementation | No | Yes | No (uses Hibernate underneath) |
| Boilerplate Code | Moderate | High | Very low |
| Transactions | Manual via *EntityManager* | Manual via Session | Auto-managed with @Transactional |
| Query Methods | JPQL/Criteria API | HQL/Criteria API | Auto-generated + custom with @Query |
| Common Use Case | Base API standard | Full control ORM | Rapid development in Spring applications |