**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**

* Go to <https://start.spring.io/>
* Change Group as “com.cognizant”
* Change Artifact Id as “orm-learn”
* In Options > Description enter "Demo project for Spring Data JPA and Hibernate"
* Click on menu and select "Spring Boot DevTools", "Spring Data JPA" and "MySQL Driver"
* Click Generate and download the project as zip
* Extract the zip in root folder to Eclipse Workspace
* Import the project in Eclipse "File > Import > Maven > Existing Maven Projects > Click Browse and select extracted folder > Finish"
* Create a new schema "ormlearn" in MySQL database. Execute the following commands to open MySQL client and create schema.

> mysql -u root -p

mysql> create schema ormlearn;

* In orm-learn Eclipse project, open src/main/resources/application.properties and include the below database and log configuration.

# Spring Framework and application log

logging.level.org.springframework=info

logging.level.com.cognizant=debug

# Hibernate logs for displaying executed SQL, input and output

logging.level.org.hibernate.SQL=trace

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

# Log pattern

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

# Database configuration

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

# Hibernate configuration

spring.jpa.hibernate.ddl-auto=validate

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

* Build the project using ‘mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com -Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456’ command in command line
* Include logs for verifying if main() method is called.

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

  LOGGER.info("Inside main");

}

* Execute the OrmLearnApplication and check in log if main method is called.

SME to walk through the following aspects related to the project created:

1. src/main/java - Folder with application code
2. src/main/resources - Folder for application configuration
3. src/test/java - Folder with code for testing the application
4. OrmLearnApplication.java - Walkthrough the main() method.
5. Purpose of @SpringBootApplication annotation
6. pom.xml
   1. Walkthrough all the configuration defined in XML file
   2. Open 'Dependency Hierarchy' and show the dependency tree.

**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');

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

* Open Eclipse with orm-learn project
* Create new package com.cognizant.orm-learn.model
* Create Country.java, then generate getters, setters and toString() methods.
* Include @Entity and @Table at class level
* Include @Column annotations in each getter method specifying the column name.

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()

}

*Notes:*

* @Entity is an indicator to Spring Data JPA that it is an entity class for the application
* @Table helps in defining the mapping database table
* @Id helps is defining the primary key
* @Column helps in defining the mapping table column

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

* Create new package com.cognizant.orm-learn.repository
* Create new interface named CountryRepository that extends JpaRepository<Country, String>
* Define @Repository annotation at class level

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**

* Create new package com.cognizant.orm-learn.service
* Create new class CountryService
* Include @Service annotation at class level
* Autowire CountryRepository in CountryService
* Include new method getAllCountries() method that returns a list of countries.
* Include @Transactional annotation for this method
* In getAllCountries() method invoke countryRepository.findAll() method and return the result

**Testing in OrmLearnApplication.java**

* Include a static reference to CountryService in OrmLearnApplication class

private static CountryService countryService;

* Define a test method to get all countries from service.

    private static void testGetAllCountries() {

        LOGGER.info("Start");

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

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

        LOGGER.info("End");

    }

* Modify SpringApplication.run() invocation to set the application context and the CountryService reference from the application context.

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

        countryService = context.getBean(CountryService.class);

        testGetAllCountries();

* Execute main method to check if data from ormlearn database is retrieved.

 **Name**: orm-learn

 **Type**: Maven Project

 **Packaging**: Jar

 **Java Version**: (e.g. 11 or 17)

 **Group**: com.cognizant

 **Artifact**: orm-learn

 **Description**: Demo project for Spring Data JPA and Hibernate

MySql Query

CREATE SCHEMA ormlearn;

USE ormlearn;

CREATE TABLE country (

co\_code VARCHAR(2) PRIMARY KEY,

co\_name VARCHAR(50)

);

INSERT INTO country VALUES ('IN','India'),('US','United States of America');

src/main/resources/application.properties

# 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

# DataSource

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

# Hibernate

spring.jpa.hibernate.ddl-auto=validate

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

**OrmApplication.java**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.ApplicationContext;

@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.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name="country")

public class Country {

@Id

@Column(name="co\_code")

private String code;

@Column(name="co\_name")

private String name;

// getters, setters, toString()

}

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> { }

package com.cognizant.ormlearn.service;

import java.util.List;

import javax.transaction.Transactional;

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

import org.springframework.stereotype.Service;

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();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

**Difference between JPA, Hibernate and Spring Data JPA**   
  
Java Persistence API (JPA)

* JSR 338 Specification for persisting, reading and managing data from Java objects
* Does not contain concrete implementation of the specification
* Hibernate is one of the implementation of JPA

Hibernate

* ORM Tool that implements JPA

Spring Data JPA

* Does not have JPA implementation, but reduces boiler plate code
* This is another level of abstraction over JPA implementation provider like Hibernate
* Manages transactions

**Refer code snippets below on how the code compares between Hibernate and Spring Data JPA  
Hibernate**

   /\* Method to CREATE an employee in the database \*/

   public Integer addEmployee(Employee employee){

      Session session = factory.openSession();

      Transaction tx = null;

      Integer employeeID = null;

      try {

         tx = session.beginTransaction();

         employeeID = (Integer) session.save(employee);

         tx.commit();

      } catch (HibernateException e) {

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

         e.printStackTrace();

      } finally {

         session.close();

      }

      return employeeID;

   }

**Spring Data JPA**  
EmployeeRespository.java

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

EmployeeService.java

@Autowire

  private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

  employeeRepository.save(employee);

  }

​​​​​​​   
  
**Reference Links:**   
<https://dzone.com/articles/what-is-the-difference-between-hibernate-and-sprin-1>   
<https://www.javaworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html>

**1. What Each Layer Is**

| **Layer** | **Description** |
| --- | --- |
| **JPA** | Java Persistence API – a **specification** (JSR 338) defining a set of interfaces and annotations (EntityManager, @Entity, etc.) for ORM and object ↔ relational mapping in Java. |
| **Hibernate** | A **concrete implementation** of JPA (and also offers its own native API). It’s a full‑featured ORM framework with caching, batching, interceptors, multi‑tenant support, and more. |
| **Spring Data JPA** | A **Spring‑provided abstraction** on top of any JPA provider (e.g. Hibernate, EclipseLink). It auto‑implements repository interfaces, handles transactions, paging/sorting, and reduces boilerplate. |

**2. Specification vs. Implementation vs. Abstraction**

1. **JPA (Spec only)**
   * Provides no code—just contracts.
   * You write code *against* those interfaces (e.g. EntityManager.persist()).
   * If you need standard portability across multiple JPA providers (Hibernate, EclipseLink, OpenJPA), you code to JPA.
2. **Hibernate (JPA + extensions)**
   * Implements JPA interfaces under the hood.
   * Exposes its own APIs (e.g. SessionFactory, HQL, native Criteria API).
   * Adds advanced features like second‑level cache, query plan caching, multi‑tenant, bytecode enhancement, and proprietary performance hints.
3. **Spring Data JPA (Abstraction)**
   * Builds on top of JPA (so you still need a JPA provider on the classpath).
   * Provides JpaRepository<T,ID>, PagingAndSortingRepository, query‑derivation from method names, @Query for custom JPQL/SQL, and built‑in paging/sorting.
   * Manages transactions transparently (just annotate your service layer with @Transactional).
   * Translates low‑level PersistenceException/HibernateException into Spring’s DataAccessException hierarchy.

**3. Configuration & Bootstrapping**

| **Aspect** | **JPA (standalone)** | **Hibernate (native)** | **Spring Data JPA (with Spring Boot)** |
| --- | --- | --- | --- |
| **Bootstrap** | Persistence.createEntityManagerFactory("unitName") | new Configuration().configure().buildSessionFactory() | Auto‑configured via spring-boot-starter-data-jpa; scans entities/packages |
| **XML vs. Properties** | persistence.xml or Java SE PersistenceUnitInfo | hibernate.cfg.xml or direct Configuration#setProperty(...) | application.properties / application.yml with spring.datasource.\* and spring.jpa.\* |
| **Transaction manager** | EntityTransaction (Java SE) or JTA in Java EE | Spring’s HibernateTransactionManager or native JTA | Auto‑wired PlatformTransactionManager; annotate services with @Transactional |
| **Exception handling** | JPA throws PersistenceException | HibernateException | Spring converts to DataAccessException |

**4. API & Boilerplate**

**4.1. Using “Plain” JPA (no Spring)**

java

CopyEdit

EntityManagerFactory emf = Persistence.createEntityManagerFactory("myUnit");

EntityManager em = emf.createEntityManager();

EntityTransaction tx = em.getTransaction();

try {

tx.begin();

em.persist(new Country("IN", "India"));

tx.commit();

} catch (Exception e) {

if (tx != null && tx.isActive()) tx.rollback();

throw e;

} finally {

em.close();

}

* You must manually open/close the EntityManager and begin/commit/rollback transactions.
* Querying uses JPQL or Criteria API.

**4.2. Native Hibernate API**

java

CopyEdit

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

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

try {

session.save(new Employee("Alice"));

tx.commit();

} catch (Exception e) {

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

} finally {

session.close();

}

* Similar boilerplate to JPA—but using Hibernate’s Session and its extensions (e.g. session.byNaturalId(), session.enableFilter()).

**4.3. Spring Data JPA**

java

CopyEdit

// 1) Repository

public interface CountryRepository extends JpaRepository<Country, String> {}

// 2) Service

@Service

public class CountryService {

@Autowired

private CountryRepository repository;

@Transactional

public Country create(Country c) {

return repository.save(c);

}

public List<Country> all() {

return repository.findAll(); // auto‑impl by Spring

}

}

* **Zero DAO implementation**: Spring Data JPA inspects your CountryRepository interface at startup, generates a proxy class that implements CRUD, paging, sorting, and query methods.
* **Transactions**: simply annotate your service method with @Transactional; Spring does the rest.

**5. Querying & Paging**

| **Feature** | **JPA** | **Hibernate** | **Spring Data JPA** |
| --- | --- | --- | --- |
| **JPQL** | em.createQuery("SELECT c FROM Country c") | Same | Supported via @Query("…") |
| **Criteria API** | JPA CriteriaBuilder / CriteriaQuery | Hibernate’s Criteria (pre‑JPA) | Expose EntityManager if needed, but often unnecessary |
| **Method‑name queries** | N/A | N/A | findByName(String name), findTop5ByOrderByNameDesc() |
| **Paging & sorting** | Manual: setFirstResult(), setMaxResults() | Same | Pageable parameter in repo methods |
| **Native SQL** | em.createNativeQuery("…") | session.createNativeQuery("…") | @Query(value = "…", nativeQuery = true) |

**6. Advanced Features**

| **Area** | **Hibernate Extensions** | **Spring Data JPA** |
| --- | --- | --- |
| **Caching** | 1st‑level (session) & 2nd‑level (configurable; EHCache, Infinispan, etc.) | You still configure at provider level; Spring Data doesn’t itself cache. |
| **Multi‑tenancy** | Built‑in support (DISCRIMINATOR, SCHEMA, DATABASE) | Expose provider config, can integrate via Spring’s AbstractRoutingDataSource. |
| **Interceptors & Events** | EmptyInterceptor, event listeners | You can register Hibernate event listeners via Spring config. |
| **Batching & Fetching** | StatelessSession, @BatchSize, @Fetch(FetchMode.SUBSELECT) | Leverage Hibernate annotations directly on entities. |

**7. When to Choose What**

* **JPA (pure spec)**  
  – You need vendor neutrality; deploying on multiple Java EE servers.  
  – You want full control over bootstrap and lifecycle in non‑Spring environments.
* **Hibernate**  
  – You need advanced ORM features (2nd level cache, multi‑tenant, custom batching).  
  – You’re already comfortable with its native APIs and want maximum performance tuning.
* **Spring Data JPA**  
  – You’re building a Spring (Boot) application and want to eliminate almost all DAO boilerplate.  
  – You need built‑in paging, sorting, query‑by‑method‑name, and easy transaction demarcation.  
  – You want Spring’s exception translation and integration with the rest of the Spring ecosystem.

**Summary**

* **JPA** is your **contract**—it tells you *what* every ORM must do.
* **Hibernate** is your powerful **engine**—it does the work and adds extra horsepower.
* **Spring Data JPA** is your **ceo**—it delegates to Hibernate (or any provider), cutting out the paperwork (boilerplate) and orchestrating transactions, queries, and Spring integration.