**Difference Between JPA , Hibernate & Spring JPA**

**JPA (Java Persistence API)**

* JPA is a specification (interface) for object-relational mapping (ORM) in Java.
* It defines a standard for persisting Java objects into relational databases.
* JPA is just an API, not an implementation.
* You need an implementation to use JPA — e.g., Hibernate, EclipseLink.

**Code Example:**

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Persistence;

import javax.persistence.EntityManager;

@Entity

public class User {

@Id

private int id;

private String name;

// getters and setters

}

// usage

EntityManager em = Persistence.createEntityManagerFactory("myPU").createEntityManager();

em.getTransaction().begin();

User user = new User();

user.setId(1);

user.setName("John");

em.persist(user);

em.getTransaction().commit();

em.close();

**Hibernate**

* Hibernate is a popular implementation of the JPA specification.
* It also adds its own features beyond JPA (like caching, HQL, etc.).
* You can use Hibernate as:
  + A standalone ORM with its own API.
  + Or as a JPA provider.

**Code Example:**

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

public class HibernateExample {

public static void main(String[] args) {

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

Session session = factory.openSession();

session.beginTransaction();

User user = new User();

user.setId(2);

user.setName("Jane");

session.save(user);

session.getTransaction().commit();

session.close();

factory.close();

}

}

**Spring Data JPA**

* Spring Data JPA is a Spring-based abstraction over JPA.
* It simplifies JPA usage by providing repository interfaces.
* You can perform CRUD and queries with minimal boilerplate code.

**Code Example:**

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

import org.springframework.stereotype.Repository;

import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

public class User {

@Id

private int id;

private String name;

// getters and setters

}

@Repository

public interface UserRepository extends JpaRepository<User, Integer> {

}

// Usage in a Spring Boot Application

@Autowired

private UserRepository userRepository;

public void saveUser() {

User user = new User();

user.setId(3);

user.setName("Mike");

userRepository.save(user);

}

**Difference Table:-**

| **Feature** | **JPA** | **Hibernate** | **Spring Data JPA** |
| --- | --- | --- | --- |
| **Type** | Specification (API only) | Implementation (of JPA) & ORM | Abstraction layer over JPA |
| **What it provides?** | Standard interfaces for ORM | Concrete implementation & extra features (HQL, caching) | Repository pattern with less boilerplate |
| **Boilerplate code** | Medium | Medium | Very low |
| **Ease of use** | Requires manual setup & config | Slightly easier with its own API | Easiest to use, auto-configures |
| **Standalone?** | No — needs provider like Hibernate | Yes — can work on its own | No — needs JPA implementation underneath |
| **Extra features** | None — only standard | Adds features beyond JPA | Adds Spring ecosystem support, auto-query generation |
| **Learning curve** | Moderate | Moderate | Low |
| **Example use case** | When you need portability & standardization | When you want JPA + extra power | When you build Spring Boot apps |