**Spring Data JPA with Spring Boot, Hibernate**

**Spring Data JPA Quick Example**

#### **1. JpaApplication.java**

* Annotated with @SpringBootApplication.
* Implements CommandLineRunner to run code at application startup.
* Creates a Student object and saves it using studentService.saveStudent(student).

#### **2. Student.java**

* Annotated with @Entity and @Table(name = "student\_1").
* Uses @Id and @GeneratedValue for primary key.
* Has a no-args and all-args constructor along with getters and setters.

#### **3. StudentRepository.java**

* Extends JpaRepository<Student, Long>, so all basic CRUD operations are inherited.
* No extra code is needed unless custom queries are required.

#### **4. StudentService.java**

* Annotated with @Service.
* Injects StudentRepository using @Autowired.
* Provides a saveStudent method to interact with the database.

**Student class**

package com.infosys.jpa;  
  
import jakarta.persistence.\*;  
  
@Entity  
@Table(name = "student\_1")  
public class Student {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private long sid;  
  
 private String name;  
 private String city;  
  
 public Student() {}  
  
 public Student(long sid, String name, String city) {  
 this.sid = sid;  
 this.name = name;  
 this.city = city;  
 }  
  
 public long getSid() {  
 return sid;  
 }  
  
 public void setSid(long sid) {  
 this.sid = sid;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getCity() {  
 return city;  
 }  
  
 public void setCity(String city) {  
 this.city = city;  
 }  
}

**JpaRepository**

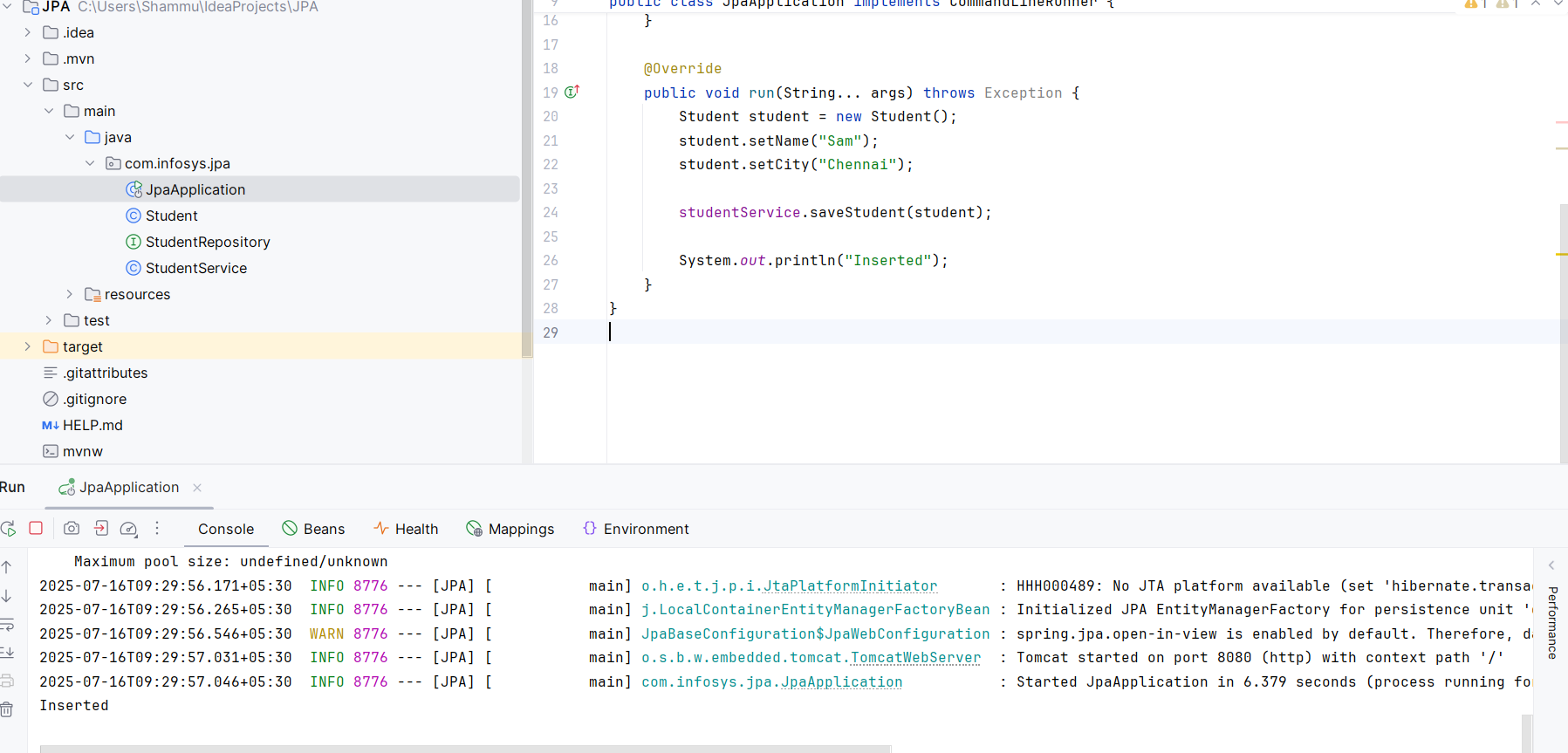
package com.infosys.jpa;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
  
public interface StudentRepository extends JpaRepository<Student, Long> {  
}

**StudentService**

package com.infosys.jpa;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service  
public class StudentService {  
  
 @Autowired  
 private StudentRepository studentRepository;  
  
 public Student saveStudent(Student student) {  
 return studentRepository.save(student);  
 }  
}

**Main class**

package com.infosys.jpa;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class JpaApplication implements CommandLineRunner {  
  
 @Autowired  
 private StudentService studentService;  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(JpaApplication.class, args);  
 }  
  
 @Override  
 public void run(String... args) throws Exception {  
 Student student = new Student();  
 student.setName("Sam");  
 student.setCity("Chennai");  
  
 studentService.saveStudent(student);  
  
 System.*out*.println("Inserted");  
 }  
}



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

**Introduction:**

In Java-based enterprise applications, interacting with databases efficiently and in a maintainable way is critical. Object Relational Mapping (ORM) frameworks bridge the gap between object-oriented design and relational databases. Three key components in the Java ecosystem for data persistence are **JPA (Java Persistence API)**, **Hibernate**, and **Spring Data JPA**. Understanding their differences and how they complement each other is essential for making informed architectural decisions.

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

JPA is a **specification** provided by Java for persisting, retrieving, and managing data between Java objects and relational databases. It provides a standard approach for ORM without being tied to any specific implementation.

* **Nature**: Interface-level specification.
* **Defined By**: Oracle (part of Java EE, now Jakarta EE).
* **Responsibilities**: Defines annotations, entity management, querying via JPQL (Java Persistence Query Language), transaction management, and object-relational mappings.
* **Needs an implementation** such as Hibernate, EclipseLink, or OpenJPA to work.

**2. Hibernate**

Hibernate is the **most widely used implementation** of the JPA specification, but it also offers additional features not included in JPA.

* **Nature**: ORM framework and JPA provider.
* **Supports JPA**: Fully compatible with JPA but also has proprietary features like Criteria API, caching, custom data types, and native SQL queries.
* **Flexible**: Can be used with or without JPA.
* **Advanced features**: Lazy loading, cascading, batch processing, second-level cache, interceptors, and custom dialects.

**3. Spring Data JPA**

Spring Data JPA is a **Spring-based abstraction** built on top of JPA and typically uses Hibernate under the hood. It drastically **reduces boilerplate code** by providing ready-made implementations for common persistence operations.

* **Nature**: Framework built on top of JPA.
* **Part of**: Spring Data project.
* **Focus**: Simplify data access and repository patterns in Spring applications.
* **Key Advantage**: Enables the creation of repository interfaces by defining method signatures; Spring will auto-implement these interfaces at runtime.

### **Comparison Table**

| **Feature** | **JPA** | **Hibernate** | **Spring Data JPA** |
| --- | --- | --- | --- |
| Type | Specification | ORM Framework (JPA Provider) | Framework built on JPA |
| Purpose | Define standard ORM mappings in Java | Provide ORM implementation | Simplify data access in Spring apps |
| Part of | Java EE / Jakarta EE | Independent / Can be part of Spring or Java EE | Spring Framework |
| Provides Implementation | No | Yes | No (delegates to JPA provider like Hibernate) |
| Entity Mapping | Yes | Yes | Yes (via JPA) |
| JPQL Support | Yes | Yes | Yes (via JPA) |
| Native SQL Support | Limited | Full support | Yes (through custom queries) |
| Custom Features | No | Yes (e.g., caching, batch processing) | No (uses underlying JPA provider) |
| Query Language | JPQL | JPQL, HQL, Criteria, Native SQL | Derived queries, JPQL, native SQL |
| Query DSL | No | Criteria API | Spring Data Query DSL |
| Ease of Use | Moderate | Moderate to Advanced | Very High |
| Boilerplate Code | Medium | Low to Medium | Very Low (minimal configuration) |
| Spring Integration | Manual | Manual or via Spring ORM | Seamless |
| Repository Abstraction | No | No (manual DAO creation) | Yes (auto-generated repositories) |
| Caching | No | Yes (first and second-level caching) | Inherits from Hibernate |
| Transaction Management | Via JTA or EntityManager | Built-in or via Spring | Managed via Spring |
| Learning Curve | Moderate | Steeper due to custom APIs | Easy (especially in Spring Boot) |
| Popular Use Cases | Standardized ORM needs | Rich ORM features and advanced configurations | Rapid Spring application development |

**Conclusion**

* **JPA** defines the standard contract for ORM in Java applications.
* **Hibernate** implements the JPA specification and adds powerful ORM features beyond the specification.
* **Spring Data JPA** simplifies and abstracts the persistence layer using JPA (and Hibernate), making it ideal for rapid application development with Spring Boot.

Choosing the right one depends on the project needs:

* For full control and advanced ORM features, use **Hibernate** directly.
* For standardization and vendor independence, use **JPA**.
* For rapid development with minimal code and Spring integration, choose **Spring Data JPA**.