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

Understanding JPA, Hibernate, and Spring Data JPA is crucial for modern Java development, especially when dealing with relational databases. They represent different layers of abstraction, each simplifying data persistence in its own way.

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

* **Role:** JPA is a **specification (JSR 338)** for persisting, reading, and managing data from Java objects to a relational database.
* **Nature:** It is purely an **API (Application Programming Interface)**. It defines a set of interfaces, annotations, and an object-relational mapping (ORM) paradigm.
* **Implementation:** JPA **does not contain a concrete implementation** of the specification itself. It's like a blueprint or a contract. Any vendor can provide an implementation that adheres to this specification.
* **Purpose:** To provide a standard way to map Java objects to database tables, allowing developers to write database-agnostic persistence code. This means you can switch JPA providers (like Hibernate, EclipseLink, etc.) without changing your core application code, as long as they adhere to the JPA standard.

### **2. Hibernate**

* **Role:** Hibernate is an **ORM (Object-Relational Mapping) tool** that **implements the JPA specification**.
* **Nature:** It is a **concrete implementation** of the JPA interfaces and provides the actual runtime code to perform persistence operations.
* **Functionality:** Beyond implementing JPA, Hibernate also offers its own set of features (often called "Hibernate-specific" or "proprietary" features) that go beyond the standard JPA capabilities. These might include custom caching strategies, specific query language extensions (HQL), etc.
* **Direct Usage:** You can use Hibernate directly as an ORM tool without JPA, but since JPA became standard, it's more common to use Hibernate as a JPA provider.

**Hibernate Code Snippet Analysis:**

/\* Method to CREATE an employee in the database \*/ public Integer addEmployee(Employee employee){ Session session = factory.openSession(); // Get a session from the SessionFactory Transaction tx = null; Integer employeeID = null;

try {  
 tx = session.beginTransaction(); // Manually begin a transaction  
 employeeID = (Integer) session.save(employee); // Save the employee object  
 tx.commit(); // Manually commit the transaction  
} catch (HibernateException e) {  
 if (tx != null) tx.rollback(); // Manually rollback on error  
 e.printStackTrace();  
} finally {  
 session.close(); // Manually close the session  
}  
return employeeID;

}

**Observations:**

* **Manual Session Management:** You explicitly open and close a Session.
* **Manual Transaction Management:** You manually beginTransaction(), commit(), and rollback() transactions.
* **Direct API Calls:** You directly interact with Hibernate's Session API (session.save()).
* **Boilerplate Code:** There's a significant amount of boilerplate code (try-catch-finally blocks, session management, transaction management) required for even a simple operation like saving an entity.

### **3. Spring Data JPA**

* **Role:** Spring Data JPA is **not a JPA implementation** itself. Instead, it is **another level of abstraction over JPA implementation providers like Hibernate.**
* **Nature:** It is part of the larger Spring Data project and provides a **repository abstraction**. It significantly **reduces boilerplate code** associated with implementing data access layers.
* **Functionality:**
  + **Automatic Repository Implementation:** By simply declaring an interface that extends JpaRepository (or other Spring Data interfaces), Spring Data JPA automatically provides common CRUD (Create, Read, Update, Delete) operations without you writing any implementation code.
  + **Query Derivation:** It can automatically derive queries from method names (e.g., findByLastNameAndFirstName()).
  + **Transaction Management:** While JpaRepository itself doesn't directly manage transactions, Spring's @Transactional annotation (which works seamlessly with Spring Data JPA) handles transaction management declaratively, removing the need for manual beginTransaction, commit, and rollback.
  + **Integration:** It seamlessly integrates JPA (and its underlying implementation like Hibernate) into the Spring ecosystem, leveraging Spring's dependency injection and transaction management capabilities.

**Spring Data JPA Code Snippet Analysis:**

**EmployeeRepository.java**

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

**// Spring Data JPA automatically provides methods like save(), findById(), findAll(), delete() etc.**

**// You can also define custom query methods here, e.g., List<Employee> findByLastName(String lastName);**

**}**

EmployeeService.java

@Service // Marks this as a Spring service bean

public class EmployeeService {

@Autowired // Spring automatically injects an instance of EmployeeRepository

private EmployeeRepository employeeRepository;

@Transactional // Spring manages the transaction for this method

public void addEmployee(Employee employee) {

employeeRepository.save(employee); // Calls the save method provided by JpaRepository

}

}

**Observations:**

* **No Repository Implementation:** You don't write any class to implement EmployeeRepository. Spring Data JPA generates the implementation at runtime.
* **Declarative Transaction Management:** The @Transactional annotation handles all the transaction boilerplate (beginning, committing, rolling back).
* **Dependency Injection:** @Autowired (a Spring annotation) handles injecting the EmployeeRepository instance into the EmployeeService.
* **Minimal Code:** The code for saving an employee is incredibly concise. You focus on *what* you want to do (save) rather than *how* to do it (session, transaction management).

### **Summary Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature/Aspect | JPA | Hibernate | Spring Data JPA |
| **Role** | Specification/API | JPA Implementation / ORM Tool | Abstraction Layer over JPA Providers |
| **Nature** | Defines interfaces and annotations | Provides concrete implementation (library) | Provides repository abstraction (library) |
| **Boilerplate** | High (if implemented manually) | High (manual session/transaction) | **Low (removes most boilerplate)** |
| **Transaction Mgmt.** | Standard API for programmatic control | Manual beginTransaction/commit/rollback | **Declarative (@Transactional)** |
| **Session Mgmt.** | Standard API for programmatic control | Manual openSession/close | **Managed by Spring** |
| **Querying** | JPQL, Criteria API | HQL, Criteria API (and JPQL, Criteria API) | Derived Queries, @Query, JPQL, Criteria API |
| **Dependency** | Javax/Jakarta Persistence API | JPA API + Hibernate Core | Spring Data JPA + JPA Provider (Hibernate) |

In essence:

* **JPA** is the **standard**.
* **Hibernate** is the **engine** that fulfills that standard (and adds more).
* **Spring Data JPA** is the **productivity layer** that sits on top, making it incredibly easy and concise to use JPA (and Hibernate) within a Spring application by removing tedious boilerplate code.