# Spring Hands-on 4: JPA vs Hibernate vs Spring Data JPA

# 1. Introduction

In this hands-on exercise, we explore the distinctions between Java Persistence API (JPA), Hibernate, and Spring Data JPA. This document consolidates the conceptual differences and includes annotated code examples to demonstrate how traditional ORM (Object Relational Mapping) compares to modern Spring Data JPA approaches.  
  
This hands-on proof also demonstrates how Spring Data JPA simplifies persistence layer implementation, reduces boilerplate, and improves readability using annotations and repositories.

# 2. Conceptual Comparison

Here is a side-by-side comparison of the three technologies:

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | JPA | Hibernate | Spring Data JPA |
| Definition | Java Specification for ORM | ORM tool that implements JPA | Abstraction layer over JPA/Hibernate |
| Type | API Specification | Library / Framework | Spring Project Module |
| Implementation Provided? | No | Yes | No (uses JPA providers) |
| Boilerplate Code | High | Medium | Low |
| Transaction Handling | Manual or via EJB | Manual via Transaction API | Automatic with @Transactional |
| Configuration | persistence.xml | hibernate.cfg.xml | application.properties/yml |
| Query Language | JPQL | HQL + JPQL | JPQL + Derived Query Methods |
| Ease of Testing | Complex | Moderate | Easy with Spring Boot Support |
| Spring Boot Support | Manual Integration Required | Requires Adapter | Full Support |

# 3. Traditional Hibernate Example

Below is a standard Hibernate-based DAO implementation to insert an employee into 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;  
}

As seen above, developers are responsible for opening sessions, handling transactions, managing exceptions, and ensuring proper closure of resources.

# 4. Spring Data JPA Example

Now let’s compare this with a Spring Data JPA implementation of the same logic.

// EmployeeRepository.java  
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {}  
  
// EmployeeService.java  
@Service  
public class EmployeeService {  
 @Autowired  
 private EmployeeRepository employeeRepository;  
  
 @Transactional  
 public void addEmployee(Employee employee) {  
 employeeRepository.save(employee);  
 }  
}

This approach dramatically reduces the amount of code needed, thanks to Spring's dependency injection and built-in transaction management. Spring Boot handles the lifecycle and configuration of the underlying persistence provider (e.g., Hibernate) automatically.

# 5. Summary and Reflection

Through this exercise, it becomes evident that Spring Data JPA provides a cleaner, more maintainable, and scalable approach to data persistence in Java enterprise applications. Unlike raw Hibernate or JPA, Spring Data JPA requires minimal configuration and allows us to focus more on business logic rather than boilerplate setup.

This document serves as proof that the concepts have been understood and implemented practically using Spring Boot 3.5.3, Java 24, Hibernate ORM 6.4.4, and MySQL 8 as the database backend.  
  
  
  
**Suriya Prasaad S**  
Email: [suriyaprasaadjayasugumar04@gmail.com](mailto:suriyaprasaadjayasugumar04@gmail.com)  
Superset ID: *6432243*  
[LinkedIn](https://www.linkedin.com/in/suriyaprasaad/)  
[Portfolio](https://suriyaprasaad.netlify.app/)