**Hands-On 4: Understanding the Difference Between JPA, Hibernate, and Spring Data JPA**

***Introduction***

In enterprise Java development, developers often come across terms like JPA, Hibernate, and Spring Data JPA. While they are related, they serve different roles. In this hands-on, we’ learnt to break down their differences using real-world context and walked through two code snippets from the exercise to highlight how the development experience changes across them.

***JPA - Java Persistence API***

JPA is essentially a standard or specification for accessing, persisting, and managing data between Java objects and relational databases. Think of JPA as a contract — it defines **what** should be done, but not **how** it’s done.

We can annotate Java classes with JPA annotations like @Entity, @Id, and use EntityManager for operations. But JPA itself doesn’t do anything unless we plug in a provider that implements it — like Hibernate.

***Hibernate - A JPA Implementation***

Hibernate is a concrete implementation of the JPA specification. It’s also an ORM (Object Relational Mapping) tool that existed even before JPA became popular. Hibernate allows developers to interact with the database using Java objects rather than SQL, and provides extra features like caching, lazy loading, and more.

Here’s the code snippet from the handout:

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;

}

***Explanation:***

* This is pure Hibernate, we have to manage the session lifecycle manually.
* Transactions need to be explicitly started and committed.
* We handle rollback and closing in finally blocks.

***What I learnt from this:***

* Hibernate gives a lot of control but also introduces boilerplate, which is the code we **have to write** over and over again — just to get the basic setup done, not to solve our real problem.
* For a simple save operation, we write a lot of repetitive code.
* Developers must manage sessions and handle exceptions carefully.

***Spring Data JPA - An Abstraction over JPA***

Spring Data JPA takes things a step further by abstracting most of the boilerplate code we saw in Hibernate. It works **on top of JPA**, so it still needs a JPA provider (like Hibernate) underneath. But it simplifies interaction with the database by providing built-in repositories.

Here’s the Spring Data JPA version from the handsOn:

***EmployeeRepository.java***

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

// No implementation required

}

***EmployeeService.java***

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

employeeRepository.save(employee);

}

***Explanation:***

* We don’t need to open or close sessions, spring handles that.
* We don’t need to write the save logic at all, the JpaRepository interface provides it.
* The @Transactional annotation tells Spring to wrap the method in a transaction.

***What I learnt from this:***

* Spring Data JPA drastically reduces code.
* It follows convention over configuration, this means that method names define behavior.
* Perfect for rapid development where standard CRUD operations are enough.

***Conclusion***

The takeaway from this hands-on is that while all three tools — JPA, Hibernate, and Spring Data JPA — help us persist data, they serve different purposes. JPA sets the rules. Hibernate plays by those rules but gives us more features. Spring Data JPA wraps it all up in a developer-friendly layer that saves us time and effort.

Depending on our project, team, and need for customization, we can choose the right level of abstraction.