**WEEK-3**

**SPRING DATA JPA AND HIBERNATE HANDSON**

**Superset ID: 6419740**

**Objective 1: Demonstrate Writing Hibernate Query Language (HQL) and Native Query**

In Hibernate and Spring Data JPA, we can perform queries using:

* **HQL (Hibernate Query Language)** – Object-oriented version of SQL that works with entity names and properties, not table/column names.
* **JPQL (Java Persistence Query Language)** – Similar to HQL; it’s a standardized query language defined by JPA.
* **Native Query** – Actual SQL query executed directly on the database.

**1. HQL / JPQL Basics**

* These queries are written using entity class and field names:
* @Query("SELECT e FROM Employee e WHERE e.salary > :minSalary")
* List<Employee> findEmployeesWithSalaryAbove(@Param("minSalary") double minSalary);
* "Employee" refers to the entity, not the table.
* "salary" is a field in the entity, not a DB column.
* This is portable across databases, unlike SQL.

**2. HQL with FETCH Keyword**

* @Query("SELECT d FROM Department d JOIN FETCH d.employees WHERE d.name = :deptName")
* Department findDepartmentWithEmployees(@Param("deptName") String deptName);
* The FETCH keyword is used for eager loading of associations to prevent lazy loading issues like LazyInitializationException.

**3. HQL Aggregate Functions**

* @Query("SELECT AVG(e.salary) FROM Employee e")
* Double findAverageSalary();
* Supported functions: COUNT(), SUM(), AVG(), MAX(), MIN().

**4. Native Query Example**

* Sometimes, we need raw SQL queries, especially for performance or complex DB-specific queries:
* @Query(value = "SELECT \* FROM employee WHERE salary > ?1", nativeQuery = true)
* List<Employee> findHighSalaryEmployees(double minSalary);
* Use nativeQuery = true to indicate that this is raw SQL.
* Table and column names must match the actual database schema.

**Objective 2: Explain the Need and Benefit of Criteria Query**

Sometimes, we need to build queries dynamically based on user input or application logic. Writing raw HQL/JPQL in such cases can get messy. That’s where **Criteria API** comes in.

* Dynamic query building (e.g., filters based on input fields).
* Type-safe queries at compile-time.
* No need to construct long string-based queries.
* Easily reusable and maintainable.

**Core Components of Criteria Query:**

1. **CriteriaBuilder** – Factory for creating criteria queries.
2. **CriteriaQuery<T>** – Represents the query object.
3. **Root<T>** – Defines the main entity/table being queried.
4. **TypedQuery<T>** – Final query to be executed.

**Example: Find employees with dynamic salary range**

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<Employee> query = cb.createQuery(Employee.class);

Root<Employee> root = query.from(Employee.class);

query.select(root).where(cb.between(root.get("salary"), 30000, 60000));

TypedQuery<Employee> typedQuery = entityManager.createQuery(query);

List<Employee> result = typedQuery.getResultList();

This avoids manual string construction.

Fields like "salary" are checked by the compiler, reducing runtime errors.