**WEEK-3**

**SPRING DATA JPA AND HIBERNATE HANDSON**

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

In Hibernate and Spring Data JPA, database queries can be written using:

* **HQL (Hibernate Query Language)** – A SQL-like language that operates on entity class names and fields, not database table/column names.
* **JPQL (Java Persistence Query Language)** – A standard version of HQL defined by the JPA specification.
* **Native SQL** – Raw SQL executed directly on the underlying database.

**1. HQL / JPQL Fundamentals**

Queries reference entity and field names instead of database identifiers:

@Query("SELECT e FROM Employee e WHERE e.salary > :minSalary")

List<Employee> findEmployeesWithSalaryAbove(@Param("minSalary") double minSalary);

* Here, "Employee" is the entity class, and "salary" is a property—not a column name.
* This style is database-independent and portable.

**2. HQL with FETCH for Eager Loading**

@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 to load related data eagerly, preventing issues like LazyInitializationException.

**3. HQL with Aggregate Functions**

@Query("SELECT AVG(e.salary) FROM Employee e")

Double findAverageSalary();

* Aggregate functions supported include: COUNT(), SUM(), AVG(), MAX(), and MIN().

**4. Native SQL Query Example**

When performance optimization or DB-specific logic is needed, raw SQL may be used:

@Query(value = "SELECT \* FROM employee WHERE salary > ?1", nativeQuery = true)

List<Employee> findHighSalaryEmployees(double minSalary);

Set nativeQuery = true to use actual SQL.  
Note: Use real table and column names as defined in the database schema.

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

In scenarios where queries must adapt dynamically—like filtering based on user inputs—writing long HQL or JPQL strings becomes cumbersome. **Criteria API** provides a structured way to create flexible, type-safe queries.

**Why Use Criteria API?**

* Suitable for building queries programmatically based on dynamic input.
* Ensures **compile-time type checking**, reducing runtime errors.
* More maintainable than concatenating query strings.
* Reusable across various filtering use cases.

**Main Components of a Criteria Query:**

1. **CriteriaBuilder** – A factory class used to construct query parts.
2. **CriteriaQuery<T>** – Represents the actual query object.
3. **Root<T>** – Defines the entity involved in the query.
4. **TypedQuery<T>** – Executes the built query and retrieves results.

**Example: Dynamic Query for 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 method avoids manual string creation, and the "salary" field is type-checked by the compiler, improving reliability.