**Objectives**

* **Demonstrate writing Hibernate Query Language and Native Query**
  + **HQL stands for Hibernate Query Language, JPQL stands for Java Persistence Query Language, Compare HQL and JPQL, @Query annotation, HQL fetch keyword, aggregate functions in HQL, Native Query, nativeQuery attribute**
    - **Reference - https://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/ch11.html**
    - **Features of JPA Query -** [**https://www.baeldung.com/spring-data-jpa-query**](https://www.baeldung.com/spring-data-jpa-query)

Hibernate Query Language (HQL) and Java Persistence Query Language (JPQL) are both object-oriented query languages used to query entities in a database, but they differ slightly. HQL is specific to Hibernate, while JPQL is defined by the JPA specification and is more standardized. Both allow querying based on entity classes and their properties rather than table names and columns. In Spring Data JPA, custom queries can be written using the @Query annotation on repository methods. This annotation supports both JPQL and HQL syntax and allows use of parameters, joins, conditions, and even aggregate functions like COUNT(), SUM(), MAX(), and AVG().

For example, a JPQL query might look like:

java

CopyEdit

@Query("SELECT e FROM Employee e WHERE e.department.name = :deptName")

List<Employee> findByDepartmentName(@Param("deptName") String name);

HQL also supports the fetch keyword for eager loading associations:

java

CopyEdit

@Query("SELECT d FROM Department d JOIN FETCH d.employees")

List<Department> findAllWithEmployees();

When working with complex queries or legacy SQL, **native queries** can be used. Native queries execute raw SQL statements directly against the database. They are enabled in the @Query annotation by setting nativeQuery = true:

java

CopyEdit

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

List<Employee> findEmployeesWithHighSalary(double amount);

JPQL/HQL provides a type-safe, database-independent way of querying entities, while native queries offer flexibility for optimized or DB-specific logic.

* **Explain the need and benefit of Criteria Query**
  + **Scenarios where Criteria Query helps, CriteriaBuilder, Criteria Query, Root, TypedQuery**
    - **Reference -** [**https://docs.oracle.com/javaee/6/tutorial/doc/gjrij.html**](https://docs.oracle.com/javaee/6/tutorial/doc/gjrij.html)

The **Criteria Query API** in JPA is a type-safe, programmatic way to construct queries dynamically at runtime. It is especially useful in scenarios where queries need to be built based on user input or varying search parameters. Unlike JPQL or HQL, which use string-based queries, Criteria API avoids syntax errors by offering **compile-time checking** and better refactoring support through Java code. This is particularly beneficial when building complex queries involving dynamic conditions, joins, and filters.

The process begins with the CriteriaBuilder, which is obtained from the EntityManager. It acts as a factory for various query components. The CriteriaQuery object defines the structure of the query, such as the type of result expected. The Root represents the main entity involved in the query, similar to the FROM clause in SQL. To execute the query, a TypedQuery is created from the EntityManager using the CriteriaQuery object, and its result is returned using getResultList() or getSingleResult().

For example, to fetch employees with a certain department dynamically:

java

CopyEdit

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

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

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

cq.select(root).where(cb.equal(root.get("department"), "IT"));

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

List<Employee> results = query.getResultList();

The Criteria API is ideal when building flexible, reusable query logic or when avoiding hardcoded string queries. It enhances maintainability, readability, and reduces the risk of runtime errors.